Technische Universität München
Lehrstuhl III – Datenbanksysteme

Adaptive Data Processing in Embedded Networks

Diplom-Informatiker Univ.
Andreas Scholz

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Die Dissertation wurde am 08.02.2011 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 05.07.2011 angenommen.

Abstract

The availability of embedded networks consisting of interconnected microcontrollers will change the way automation systems are built in the future. Instead of hierarchical systems with a centralized control logic, we will see an increasing deployment of "smart" nodes on the field level. These nodes can be used to execute applications directly in the embedded network, thereby reducing the required network bandwidth and increasing the scalability of automation systems. We will also see an increasing number of applications that incorporate devices from both, the IT domain and the field level, e.g., real-world aware business applications with direct access to the information gathered at the field level.

Embedded networks have special characteristics that have to be taken into account during development, such as heterogeneous nodes and communication media, resource constraints imposed by the used microcontrollers and network dynamics caused by the addition or failure of nodes. These unique boundary conditions prohibit the direct application of technologies developed for IT networks and attracted researchers from various areas. A prerequisite for the widespread deployment of embedded networks is a comprehensive development platform that bundles the results of these research efforts and supports a fast and efficient development of applications. This work aims at providing the basic building blocks of such a platform and also describes a prototypical realization: the ϵSOA platform. The ϵSOA platform was tailored for building systems under the above mentioned boundary conditions. It is based on three design principles: a service oriented architecture, a data stream based execution model and a model driven development approach. We will show how these concepts can be used to create a highly customizable middleware that can scale its functionality based on application requirements. Furthermore, we present optimization techniques that allow adapting the execution of applications based on application requirements and the characteristics of the underlying network. Besides the fundamental concepts we will also present several technical solutions that provide the necessary prerequisites for these optimizations, including amongst others a modular communication layer, a service migration algorithm, a cross-layer communication interface and a service bridge that allows a seamless interaction between field level devices and Web services.

Acknowledgements

I am pleased to thank all the people that contributed to this work. First of all I want to thank Prof. Alfons Kemper for giving me the opportunity to work in this fascinating research area and his invaluable input on the various research topics tackled in this work. I also want to thank Prof. Dr. Harald Kosch from the University of Passau for volunteering to review my thesis.

My special thanks go to our cooperation partners at Siemens CT, Dr. Jörg Heuer, Anton Schmitt and Martin Winter, for the many hours of fruitful discussions. The possibility to test our implementation in an industry relevant setting and the feedback provided by these individuals had major impact on the outcomes of this work and ensured the developed solutions are applicable in an industrial setting. I want to thank Dr. Christian Buckl, Irina Gaponova and Stephan Sommer, who worked with me for several years on this topic and greatly influenced the outcomes of this work, for the productive cooperation, the inspiring discussions and the relaxed working atmosphere.

I also want to thank all my colleagues at the TUM for the feedback, support and help provided over the years. This includes Martina-Cezara Albutiu, Stefan Aulbach, Veneta Dobreva, Daniel Gmach, Prof. Dr. Torsten Grust, Benjamin Gufler, Sebastian Hagen, Dean Jacobs, Ph.D., Stefan Krompaß, Dr. Richard Kuntschke, Manuel Mayr, Fabian Prasser, Dr. Angelika Reiser, Jan Rittinger, Michael Seibold, Jessica Smejkal, Dr. Jens Teubner, Dr. Tobias Scholl and all students that worked on the implementation of the ϵSOA platform. Special thanks to our secretary, Evi Kollmann, who took care of all the tedious administrative work.

Last but not least I want to thank my family and my friends who always supported me, even during the “hard” times where I was busy writing this thesis in the evenings and on weekends.

Contents

CHAPTER 1

Introduction: Embedded Networks

Recent technological advances and continuously dropping hardware prices have paved the way for the development and installation of embedded networks. Embedded networks are composed of tiny interconnected computers with diverse processing, sensing and actuation capabilities. The widespread installation of such networked devices allows building computer based models of our environment with unprecedented accuracy. The long envisioned goal of pervasive/ubiquitous computing - the elimination of the gap between physical world surrounding us and the computer systems we use for storing and exchanging information about this world - seems at hand. Having access to up-to-date information about our surroundings anytime and anywhere and the possibility to directly control the actuation devices present in our environment dramatically changes the way we interact with computer systems and the way computer systems interact with our environment.

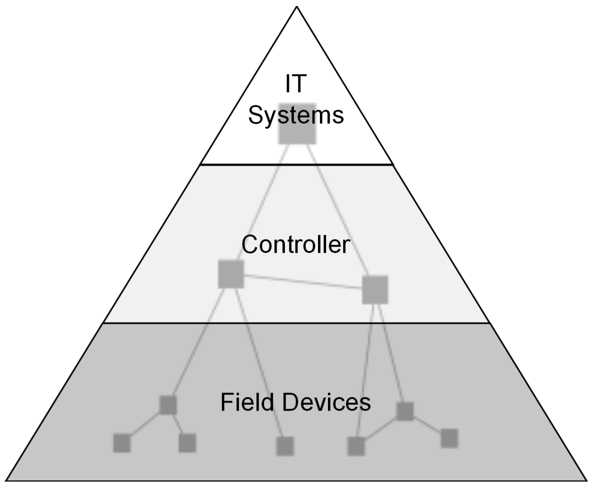
Embedded networks are emerging due to several reasons. One reason is the increasing availability of network enabled sensor devices. Consumer devices such as cell phones, navigation systems and PDAs are becoming more and more interconnected, e.g. via WLAN technology or personal area networks. At the same time, an increasing number of sensors is installed in these devices. Modern handhelds already contain a lot of built-in sensors, such as GPS modules, orientation sensors, and compasses. There are several companies and research labs working on sensor board extensions for cell phones that provide additional devices for environmental and health monitoring. This trend results in a variety of network-enabled sensor devices that can be combined to create new applications and smart systems that incorporate devices carried by the user.

A second reason is the availability of cheap sensor and actuator devices. Dropping hardware prices and improved miniaturization allows the deployment of sensor and actuator devices in new environments. An obvious application field is environmental monitoring. The more sensor devices we install in our environment, the more accurate the information we can gather is and the better the forecasts and decisions we make based on this information will be. These benefits are not limited to environmental monitoring. Modern production sites or logistic chains can be improved by adding additional sensor devices, too. The goal in these environments is to achieve *real-world awareness* in computer systems. All business relevant information, such as the status of production sites, the logistic chain or productivity rates are monitored and any changes are immediately reflected in the business back-end. A real-world aware IT infrastructure allows the timely reaction to exceptional situations in the production environment and optimizations on a fine grained level due to the availability of very detailed and up-to-date information about the production process. With dropping unit costs we will see both, an increasing number of such embedded networks and an increasing number of nodes contained in each of these networks.

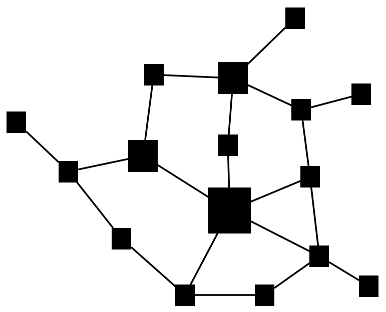
A closer look at recent standardization efforts and projects reveals another source for the emergence of embedded networks. A lot of effort is put into pushing IT technology down to control and automation systems. The driving idea is to replace domain specific solutions with well known and established IT technologies, in order to reduce costs and increase interoperability between automation systems. In the last years we already saw this trend w.r.t. physical communication media. Many protocols used in automation systems (e.g. PROFIBUS[3]) have been extended to support Ethernet (e.g. PROFINET[3]). More recently this trend is also extending to the network protocol layer. Approaches such as 6LoWPAN[63] or IP500[68] provide IP based access to field level devices. This results in a much higher interoperability between IT systems and automation systems. Sensor devices are becoming seamlessly available from the IT world and different automation systems can be interconnected. This trend also influences the architecture of automation systems. Instead of centralized architectures, e.g. star architectures comprising a central controller with attached sensor and actuator devices, these new technologies foster the development of decentralized solutions based on smart nodes that are interconnected via an IP network, hence embedded networks.

The presence of embedded networks will change the way automation systems[1] are built in the future. Traditionally, automation systems have been designed in a hierarchical way, the so called "automation pyramid". The automation pyramid is shown in Figure 1.1(a). The bottom layer contains field devices, such as sensors and

[1]This work focuses on building, process and manufacturing automation systems. Automation solutions in these and similar domains are developed individually for each customer by connecting off-the-shelf sensors, actuators and Programmable Logic Controllers (or modules composed of these devices) offered by vendors such as Siemens. Automation solutions used in modern cars or planes are another class of systems. To a certain degree, the presented solutions can be mapped to this class, too. However there are fundamental differences between both classes that have to be considered. One example is that modern cars are produced in large volumes, and the design of specialized solutions for individual product lines is possible. Another difference is that extensibility and reconfiguration features are far less important for modern cars than for modern production sites.



(a) The Automation Pyramid



(b) Converged Network of Heterogeneous Devices

Figure 1.1: Automation Systems, Present and Future

actuators. These devices are attached via field bus systems to controllers, which form the middle layer of the automation pyramid. The controllers host the application logic and periodically retrieve measurements from sensor devices in the field level and produce control signals for actuators on the field level. A direct communication between field devices is often not possible due to the used communication technology or not used. The controllers are connected to the IT backend, often using IT networks, such as Ethernet. The IT backend hosts enterprise systems, for example for production planning or order management. With the emergence of embedded networks, the separation between the layers in the pyramid will disappear. Devices at the bottom layer are getting smarter, i.e., they are no longer mere sensors or actuators that are only capable of providing raw measurements or executing simple actuation tasks. Instead we will see an increasing number of "smart" nodes, i.e., nodes with programmable microcontrollers. The processing power provided by these microcontrollers can be used to execute parts of the automation applications directly in the embedded network. The possibility to perform filtering, aggregation and control tasks directly in the embedded network reduces the required network bandwidth and greatly increases the scalability of automation systems.

When looking at the two topmost layers, we see a similar situation. The presence of a unified communication infrastructure that allows a seamless bi-directional communication between IT systems and devices in the automation domain will eliminate the boundary between the IT and the automation layer. We will see a growing number of applications that incorporate devices from the IT layer and the automation layer. One example are real-world aware IT systems with direct access to the information gathered at the field level. Another example are automation systems

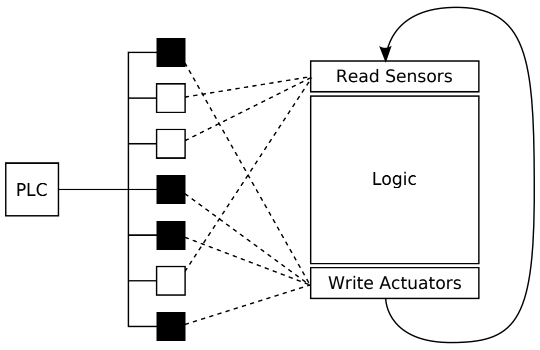


Figure 1.2: Automation System Software Architecture

that rely on information provided at the IT level for performing their tasks, such as a smart building using a weather forecast provided by an IT service or a production system using information from the enterprise back-end.

As a consequence, we will see automation systems based on an architecture like the one depicted in Figure 1.1(b). IT systems are interweaved with field devices in a single unified network[2]. Applications can access devices from both worlds and are executed in a distributed way in the embedded network. The ultimate vision is to build an Internet of Things, a converged network comprising the network of information systems we already know - the Internet - and the smart objects surrounding us.

But we still have a long way to go until we reach the Internet of Things. The widespread installation of large numbers of devices is only possible if the per unit price is low. As a consequence, the nodes in an embedded network will be built of tiny microcontrollers with very limited storage and processing capabilities. Depending on the application scenario, the nodes might even be battery powered and a wasteful use of battery resources will result in a short lifetime of nodes. The straightforward approach of simply extending the Internet to include the devices in an embedded network is therefore not possible. The limited capabilities of the nodes prohibit the direct application of the solutions we know from the Internet domain, such as Web service technologies. Instead, we have to strip down existing protocols to make them usable in the context of embedded networks and develop new solutions that are tailored to the needs of embedded networks. It is decisive that these solutions are compatible with the technologies known from the Internet domain; otherwise we will not get one converged Internet of Things but a multitude of embedded networks running in parallel to the Internet.

A side effect of the changing communication infrastructure is a paradigm shift concerning the software architecture used for implementing automation systems. Nowa-

[2]The unified network is from an application point of view. On the network and lower layers we will still see bridges and gateways that convert between the technologies used in the different domains. However these differences are transparent from a functional point of view.

days, automation solutions are commonly based on a cyclic execution model[3], for which Figure 1.2 illustrates an example. A central control unit, the Programmable Logic Controller (PLC), executes the automation software in a cyclic manner. Each cycle begins with fetching data from all sensor devices. After that, the program logic is executed. Note that even if two subsystems perform independent tasks, the application logic from both systems will be executed in a single monolithic operation on the PLC. The last step of each cycle is sending control signals to all actuator devices. It is possible to use more than one PLC in an automation system. All PLCs use a shared logical memory for storing sensor reading, intermediate results and output values. The PLCs are organized into a hierarchical structure and the control logic is split into several parts, one for each PLC. After reading the sensor values, each PLC executes its control logic and then sends the complete shared memory to all PLCs further down in the hierarchy. Due to this mechanism, the execution model is essentially the same as in the case of a single PLC.

This software design paradigm is suitable for static automation systems that are planned once and then operated unchanged for multiple years. The replacement or the addition of functionality is challenging because any modification that increases the execution time of the cycle affects the whole automation solution. Another problem is that nowadays automation solutions are often hard to debug and to maintain. There are concepts for structured and modularized software development in automation systems, such as the Function Blocks specified in IEC 61499[67] and IEC 61131[66]. Nevertheless, the code executed on a PLC is often hard to understand, even for trained engineers. The reason is that function blocks specify interfaces on the data level. A communication between different subsystems of the automation solution is performed through shared memory. These memory areas can be reused throughout the execution of the logic and it is impossible to determine what data is stored in a memory area without a detailed analysis of the code that is executed on the PLC. In the IT domain, service oriented software architectures have already shown that a decomposition of monolithic applications into loosely coupled services can greatly increase the maintainability, readability and extensibility of software systems. We will show that it is possible to build automation systems with service oriented design principles and that the benefits known from the IT domain can be transferred to the automation domain, too.

This work aims at laying the foundations for the next generation of control and automation systems based on embedded networks and service oriented design principles. We derive the requirements a system development platform for embedded networks has to fulfill and propose a framework that covers all aspects of system development, including a service oriented system architecture, an efficient middleware for nodes in an embedded network and a model driven software development approach. The solutions proposed in this work have been carefully designed to match

[3]Most systems also offer mechanisms that allow event-like processing of information parallel to the cyclic execution. Nevertheless the cyclic model is predominant, especially because timing guarantees are hard to establish using the non-cyclic execution models.

the spirit of the Internet of Things. All presented technologies are designed in a way that allows a seamless integration with Web service based solutions. We will show that it is possible to create converged embedded networks comprising resource constrained microcontrollers and Web service based IT systems. Furthermore, we show how a service oriented paradigm can be used to develop modular and extendable automation solutions that are easy to maintain and that automatically adapt the execution of applications based on the characteristics of the underlying hardware and the requirements of the executed applications. Of course this work is only a first step towards these goals. However we are confident that the presented concepts and technologies can be used as building blocks for future automation systems and will hopefully pave the way for further research directing at the Internet of Things.

We will first present a short overview of hardware platforms and operating systems for embedded networks. Throughout this work, we distinguish two classes of embedded networks: *monitoring oriented* and *control oriented* embedded networks. Both classes and their specific requirements are explained in more detail in the following sections. We will conclude this chapter with a detailed analysis of control oriented embedded networks. The requirements and properties described in this analysis motivated the creation and implementation of our development platform for embedded networks, which we will present in the main part of this work.

1.1 Hardware Platforms

Several hardware platforms have been developed for (wireless) sensor networks, e.g., the MICA platform (which is based on microcontrollers from Atmel), the TelosB/T-Mote Sky platform (which is based on microcontrollers from Texas Instruments)[4]. Another example are the nodes from ScatterWeb[5] or the WaspMotes from Libelium[6]. These architectures follow the general design depicted in Figure 1.3.

The *microcontroller* and the *communication module* are the core of every sensor node. Nodes are often also referred to as “motes” in the research community. We will use the term mote and node interchangeably throughout this work.

The microcontroller contains the *processor* (8 or 16 bit CPUs in most cases). Most microcontrollers implement a Harvard architecture, which strictly separates between the program memory (called ROM in the figure), which contains the executable code, and the data memory (called RAM in the figure), which contains the data. The ROM typically has a size of a few tens or hundreds of kilobytes, the RAM a size of 10 kilobytes or less. Many microcontrollers are shipped with an on board flash memory that allows storing comparably large data sets with some hundred kilobytes up to a few megabytes[7]. Peripheral devices such as sensor and actuator devices can be attached via an I/O interface offered by the microcontroller. The type and number

[4]Further documentation can be found at the Crossbow Homepage (http://www.xbow.com)
[5]http://scatterweb.com
[6]http://www.libelium.com/products/waspmote
[7]Some microcontrollers also offer an interface for an external flash storage.

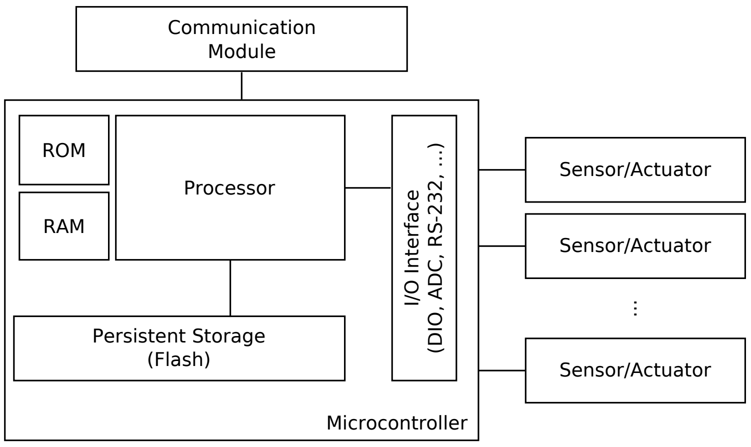


Figure 1.3: Sensor Node Hardware Architecture

of I/O ports depend on the microcontroller and range from digital I/O ports over analog I/O ports to serial interfaces, such as RS-323, and bus interfaces, such as SPI[8] or I^2C[107].

The communication module provides a communication infrastructure for nodes in the embedded network. For wireless nodes, radio modules based on the IEEE 802.15.4[56] standard have become quite popular. They provide an energy efficient alternative to the 802.11[55] based WLAN networks known from the IT domain. Other modules, wired or wireless are also possible, e.g., an Ethernet interface or an interface to a field bus system.

For our demonstrators, we used TelosB motes from Crossbow and the Java based SunSPOTS[9] from SUN (Oracle). The basic architecture of a SunSPOT is similar to the architecture depicted in Figure 1.3. SunSPOTs additionally contain a hardware based Java Virtual Machine implementation, more details about the SunSPOT platform can be found in [139]. The TelosB Mote has a ROM size of 48 kilobytes, a RAM size of 10 kilobytes and a flash memory with 1 megabyte size. Meeting these resource constraints was a design target for the solutions presented in this work.

1.2 Operating Systems

There are several operating systems available for sensor nodes. Probably the most popular ones are TinyOS[10] and Contiki[11]. We will not present both systems in detail here, a list of publications for each operating system can be found on the corresponding Web site. Both systems use an event driven execution model. Events

[8]There is no single agreed upon standard for SPI, a general overview can be found in Wikipedia
[9]For more information see `http://www.sunspotworld.com/`
[10]`http://www.tinyos.net/`
[11]`http://www.sics.se/contiki/`

can be generated by sensor/actuator devices attached to the node, by incoming network packets or by applications running on the node. Applications can register to events and process the received/measured data. Both systems contain mechanisms that allow saving energy resources by putting the node into a sleep mode if there are no pending events. Besides this core functionality, both systems offer the essential building blocks for sensor node applications, such as drivers for various hardware platforms, network protocols, or timers. Nevertheless, both systems are strictly optimized for resource constrained environments and many functionality known from PC operating systems is not available, e.g., multithreading, private memory areas, a differentiation between user and kernel mode, or a hardware abstraction layer. A difference between both systems is that Contiki contains a module that allows loading new application code at runtime. This mechanism can be used to dynamically download code to the sensor node. In TinyOS, such updates are currently not possible without rebooting the node.

1.3 Monitoring Oriented Sensor Networks

Embedded networks have been receiving growing interest in the research community in recent years and a lot of research has been done in this area. Most of this work is focused on pure sensor networks, i.e., networks which possess no actuators and perform solely measurement and observation tasks. We will use the term “monitoring oriented” sensor networks to refer to this class.

1.3.1 (Wireless) Sensor Networks

In the last years, a special subclass of sensor network has drawn a lot of attention: “wireless sensor-networks” (WSNs). WSNs use radio modules for the communication between individual nodes in the network. This introduces new complexities compared to wired communication, but also increases the flexibility for the deployment of such networks. Nodes can be placed freely and even moved at runtime[12]. A possible application scenario of wireless sensor networks is the monitoring of modern plants [152] or flexible production sites, which prohibit the use of wired solutions because of environmental influences or frequent reconfigurations. Another application scenario is environmental monitoring. A quite prominent example is the bird observation project on Great Duck Island[95]. By using WSNs, a fine grained monitoring is possible without disturbing the observed animals. The ZebraNet[73] project uses WSNs to track animals in large, wild areas. A special characteristic of this application scenario is that there is no stable network structure due to the mobility of individual nodes. Instead most communication has to be performed using ad-hoc established links whenever two tracked animals are in close proximity of each other

[12]There are some boundary conditions, such as the maximum distance between nodes or line-of-sight requirements

or are near a base station. An overview of other application scenarios for WSNs can be found in Römer et al.[123].

Wireless Sensor Networks pose several challenges. Resource constraints and limited energy resources require efficient and robust communication and data processing paradigms. Node failures, e.g., due to energy depletion or environmental influences, are common and must be compensated to guarantee a continuous operation of the sensor network. The large number of nodes encountered in sensor networks requires new management mechanisms for the efficient reconfiguration and reprogramming of whole sensor networks. Besides these topics there are a multitude of additional research directions. We will present related work for the individual topics studied in this work at the end of each section.

The nodes used in the projects mentioned above are still fairly large and expensive. An ongoing research field is to further decrease the size, energy consumption and price of sensor platforms. The ultimate goal is to design a “smart-dust” network - a sensor network comprising hundreds or even thousands of very inexpensive and very small nodes. Ideally, these nodes are small and lightweight enough to float in the air - like dust. An use case for these networks is disaster monitoring, for example by dropping nodes from a plane in order to monitor flood levels, temperatures, etc. An overview of research challenges in this area can be found in Kahn et al.[78] and Warneke et al.[166].

1.3.2 The Global Sensor Network

The increasing deployment of sensor networks has fostered the vision of “global sensor networks” (GSNs) comprising thousands or millions of geographically dispersed sensor devices. The vision of GSNs is to provide sensor information with wide (or even global) coverage through a unified easy-to-use interface to a multitude of users, even if the data stems from different underlying sensor networks. A GSN can be seen as a network of sensor networks. Research in this area has been started and aims towards providing mechanisms that allow to deal with huge numbers of devices, as encountered in the Internet of Things. The IrisNet project[44] targets a software platform that offers basic sensor network functionality, such as collecting, filtering and querying data, at a global scale. The Open Geospatial Consortium (OGC)[115] defines a set of XML standards for various measurement and observation application fields. These standards can be used as a basis for data exchange between different sensor networks. Holman et al.[53] describe a possible application scenario of a GSN using video cameras to monitor shorelines. The authors bring forward the argument that many of the nowadays installed “beach cams” could also be used for scientific or management purposes. Through the addition of image processing functionality, these cameras can be turned into an intelligent sensor network for coastal monitoring.

1.4 Control Oriented Embedded Networks

This work focuses on networks for control and automation purposes. These networks contain not only sensor devices to monitor the environment, but also actuator devices to interact with the environment (this kind of network is often also called SAN: Sensor Actuator Network). From a hardware point of view, monitoring and control oriented networks look very similar: both network types are composed of networked microcontrollers. However there are fundamental differences that require distinct solutions for both types of networks. The infrastructure of control oriented networks is comparably stable. The addition of new devices (or the failure of existing devices) is possible in control oriented networks, but these changes are rather exceptional situation. In many monitoring oriented SNs, this is not the case and network protocols and applications are often designed with mechanisms that allow compensating node failures. Another difference is that control oriented networks typically execute multiple different applications at once, which access different subsets of the available hardware. Monitoring networks on the other hand typically run a single data collection and processing application. We will present an overview of application scenarios and characteristics of control oriented embedded networks and the differences compared to monitoring oriented networks in more detail in the following sections.

It is impossible to draw a strict border between monitoring and control oriented networks. In some cases, networks with pure monitoring tasks will possess characteristics similar to control networks. Consider for example an embedded network used to monitor a production site, which possesses a well planned and fixed network structure with fairly reliable nodes. Such a network could be seen as a special case of a control oriented network that possesses no actuator devices and executes only a single monitoring application. To improve the readability, we will use the term "control oriented" network to refer to networks that possess the characteristics mentioned in the following sections. This term does not imply that the network actually performs a control task. The presented solutions may also be applied to networks executing monitoring, observation and data collection tasks - if these networks possess characteristics similar to control oriented networks.

1.4.1 Application Fields for Control Oriented Embedded Networks

Control oriented embedded networks have many application fields. Probably the most obvious one are process and production automation systems. As mentioned in the introduction, the emergence of embedded networks in this area is driven by the increasing convergence between IT and automation systems, combined with the presence of smart devices at the field level. Another application field of embedded networks is the automotive industry. Modern cars contain more and more electronic control units (ECUs) and different bus systems for the data exchange between these ECUs. In addition to these built-in devices, more and more external devices have to be integrated into the on-board network of modern cars, such as cell phones, navigation systems, multimedia players, etc. This trend is not restricted to communication inside the car but will most likely also extent to communication between the car and its environment. If vehicle-to-vehicle communication becomes available, a multitude of new sensor devices and information sources will become available and have to be integrated into next generation automobiles. A similar example are battery powered cars. The charging process - and the billing process attached to it - requires a communication between the car and the charging station, often referred to as vehicle-to-grid communication.

The power grid is another application field for embedded networks. Like in the automation domain, the power grid will be enriched with more and more smart devices. Electric cars are one source. The increasing number of solar/wind power generators installed in many households creates new challenges for the operators of energy grids. The generation of electric power is shifting from a few large power plants, to thousands of small energy generators distributed throughout the power grid[13]. In order to manage the power distribution and to ensure a stable operation, the consumption and generation of energy at individual nodes (households) in the power grid has to be measured, leading to large scale embedded networks (often called Smart Grids).

Control oriented embedded networks are also used in building and home automation scenarios. Although the technological foundations and even products for home automation systems are already available for several years, a widespread installation of automation systems by home owners is not observable. Instead, automation solutions are - with some few exceptions - limited to commercial buildings, such as office buildings, hospitals, etc. This will most likely change in the near future, as more and more homes already contain a network infrastructure for the communication between multimedia devices, PCs, etc. An automation solution that leverages this existing infrastructure could provide an affordable alternative compared to the dedicated home automation networks available nowadays. Such a home automation network could also be attached to the smart power grids described in the previous section. One vision is that future power grids will use a variable power price to reduce power peaks and smooth out the overall utilization. The power price will change based on the current demand and can be queried from the smart energy meter. Intelligent consumers in the household, such as refrigerators, washing machines, a charging station for an electric car, can schedule their power consumption in a way that minimizes the overall energy costs.

Besides the already mentioned ones, there are many additional applications fields for embedded networks. Most examples presented in this work are taken from the domain of home/building automation, because most readers should have an intuitive understanding of the requirements in this domain. Because of very similar requirements, the presented concepts can be transfered to process/production au-

[13]Nowadays the vast majority of power is still generated by large power plants. To deal with the growing number of regenerative energies resources, new management and control mechanisms have to be found, which are studied in many research projects targeting the development of the next generation of power grids.

tomation systems, too. We are confident that the core ideas can be applied in other applications fields, too - perhaps with some modifications to care for the special characteristics of the targeted system.

1.4.2 Characterization of Control Oriented Embedded Networks

Control oriented embedded networks have special characteristics and requirements that have to be supported by a development platform. We identified the following requirements, which also represent the design goals for the ϵSOA platform, our development platform for embedded networks.

1.4.2.1 Resource Constraints

An important characteristic of embedded networks are resource constraints. In order to keep the unit costs low, many nodes in embedded networks will be designed based on small microcontrollers with very limited processing power and memory resources in the order of a few tens or hundreds of kilobytes. It is foreseeable that, just like in the area of personal computers, hardware prices will drop in the future. However, dropping unit prices will not necessarily lead to nodes with increased resource capacities. In many application fields, it is more beneficial to increase the number of nodes in the embedded network instead of increasing the capacity of every single node, for example to achieve a better coverage. Additionally, a cheaper price will also open up new application fields for embedded networks that will -again- require low per-unit prices. As a consequence, the challenge of designing applications on resource constrained embedded networks will persist, even under the assumption of dropping hardware prices.

Resource constraints cannot be overcome with a single technological solution and influence many aspects of system design. They require the design of resource efficient network protocols, a lightweight execution environment on the nodes and optimization techniques that distribute workloads over multiple nodes in the network.

1.4.2.2 Heterogeneity

In contrast to monitoring oriented networks, which are often built by installing a set of similar nodes in the environment, an embedded network built for automation purposes is typically heterogeneous.

Heterogeneous Nodes In order to minimize costs and to build as energy efficient nodes as possible, control oriented networks often comprise a multitude of specialized nodes with different characteristics. On the one hand there are lightweight and cheap nodes which can be installed in large numbers, e.g., measurement nodes with attached sensor devices. These nodes often possess only very limited processing and storage capacities. On the other hand, the network will also contain more powerful nodes which provide enough resources for complex control and processing tasks. Besides the nodes directly involved in the control operations, embedded networks contain additional nodes for the management and supervision of the network. Another source of diversity are devices that are used to manage and configure the embedded network, e.g., Laptops that are dynamically attached and removed, or end user devices such as cell phones.

Multiple Communication Media A second source of heterogeneity are different communication media used inside an embedded network, ranging from field bus systems over Ethernet based networks to wireless communication media. Each of these technologies and the used network protocols have differing features and capabilities that have to be taken into account during the installation, configuration and execution of applications in the embedded network. Multiple communication technologies can even be used simultaneously in an embedded network, e.g. wireless mobile nodes which are connected to a wired backbone. The challenge is to provide a seamless and efficient communication across such heterogeneous networks.

1.4.2.3 Distributed Execution of Applications

Embedded networks require a programming paradigm that supports the distributed execution of applications. One reason is the distribution of sensor and actuator devices. Because it is unlikely that a single node possesses all required hardware devices, the execution of applications typically involves multiple nodes. The second reason are the resource limitations on the nodes. In many cases, the workload created by an application has to be distributed over multiple nodes in order to achieve a resource efficient execution and to maximize the overall lifetime of battery powered networks. A distributed execution model can also be used to remove single points of failure. If a node in an embedded networks fails, only applications that explicitly depend on functionality provided by this node, such as a specific sensor or actuator device, have to be stopped. The remaining applications will still be able to perform their tasks. Finally, applications built of distributed components also offer the possibility to control the resource utilization on a fine grained level. If a bottleneck is detected, individual application components can be moved away from the overloaded nodes in order to provide a more homogeneous utilization. Such a mechanism greatly increases the scalability of networks. If a network requires more computing resources, e.g., to resolve an overload situation or to support a new application, these can be provided by simply adding one or multiple nodes to the embedded network. There are no scalability limits such as a limited amount of I/O ports at a central controller or similar constraints.

A distributed execution is also beneficial from an optimization point of view. Often the amount of transferred data can be reduced by placing the data consuming control logic nearby the data producing sensors. Such optimizations can greatly reduce the required network traffic and help creating more stable and more efficient networks.

1.4.2.4 Simultaneous Execution of Multiple Applications

Sensing oriented embedded networks are often running only a single data collection application that collects, aggregates and stores sensor readings. In contrast to this, control oriented embedded networks typically solve multiple control and automation tasks at once. As a consequence, control oriented embedded networks have to be capable of executing multiple distributed applications simultaneously. Each of these applications may access a subset of the available sensors, actuators and processing resources. An important optimization goal is to minimize the interference between these applications. Shared resources such as communication channels or storage and processing capacities have to be allocated to the individual applications in a way that avoids overload situations.

1.4.2.5 Event Driven Processing Model

A typical task of control oriented networks is the control of actuator devices based on sensor readings. An application performing this task can be implemented efficiently following an event driven processing model. Upon the arrival of new data, e.g. new sensor readings, the execution of the application logic is triggered. Based on the outcome of this calculation, a new event is created. This event is forwarded to the actuation device. The actuator will react upon the arrival of the event by performing some actuation task. In many cases these events will be generated on a regular basis, e.g. by a sensor device that performs measurements periodically. A middleware for embedded networks should provide efficient mechanisms to manage and distribute such sequences of events. Besides periodic measurements there are also non-periodic interactions, e.g., messages for monitoring, reconfiguration or other administrative tasks. The embedded network also has to support this communication pattern.

1.4.2.6 Network Dynamics

Control networks have to be reconfigurable at run-time. At any time, new nodes with previously unknown functionality can be added and existing nodes can fail. The embedded network should be capable to adapt to such changes and re-optimize the execution of applications based on the new boundary conditions.

Not only the hardware in an embedded network is subject to change, but also the software executed on this hardware. Application fields of embedded networks can change and new applications will be added to provide new functionality. As a consequence, the purpose of individual nodes in the network is not fixed but changes throughout the lifetime of the network. This requires a life cycle management that supports the installation, startup, shutdown, and removal of individual components and whole applications in the network.

Please note that despite these dynamics, control oriented networks still possess a comparably stable network structure compared to Smart Dust networks or ad-hoc networks with mobile devices, etc. The concepts presented in this work were designed and optimized with this stability in mind.

1.4.2.7 Web Service based Interfaces

Nowadays, embedded networks do not operate in isolation but are connected to the Internet or wide-area networks and interact with external systems - sometimes with multiple different systems at the same time. The interaction can occur in both ways, embedded networks can consume data provided by external sources or supply status and monitoring information to remote systems.

One example is the integration of data supplied by Web service running in the Internet or an enterprise back-end.

1.4.2.8 End-user Programming

A key advantage of embedded networks compared to centralized control systems is the possibility to dynamically integrate new devices in existing installations. New devices can be added to the network at runtime and integrated through the installation of new applications that use the additional hardware or by updating and reconfiguring running applications. In order to fully exploit the benefits of such a system architecture, it is decisive that the integration can be performed in an easy and intuitive manner. If a comparably simple task, such as the installation of a new switch in a building automation scenario, requires complex planning and the availability of trained personnel, the benefits provided by the embedded network infrastructure will be easily outweighed by the administrative costs for performing such an operation. The goal therefore has to be to enable the users that interact with the embedded network to perform such tasks themselves.

Depending on the application field, different groups of people will perform configuration tasks in an automation system. In a home automation scenario, reconfigurations will be performed by the home owner himself, who most likely will only have domain knowledge and little to no knowledge about the technical background of their automation tasks. In a larger building such as an modern office building, trained personnel from a facility management provider will do most of the installation and maintenance tasks. In a production automation scenario, engineers will be the users that interact with the embedded network and trigger reconfigurations to optimize the performance of the production process. The common denominator of these different user groups is that, although they possess highly different knowledge concerning the technical background of their application domain, none of them will be an expert in the area of embedded network programming. An embedded network middleware should empower an end-user to do the necessary configuration and installation tasks himself. That means, the user should be able to specify what application he wants to run on the embedded network based on domain knowledge and the embedded network middleware has to ensure that the resulting application will achieve the intended goals and will operate safely on the embedded network.

1.4.2.9 Multiple User Groups

During its lifetime, multiple different user groups interact with an embedded network. One group are hardware developers that design, specify and implement nodes for a given application scenario. The second group are installers that deploy nodes in a given environment and optionally do some configuration for generic nodes. The third group are application developers that create a working embedded network out of the individual components installed by the installer. During runtime, manager will supervise the embedded network and perform adjustments and reconfigurations to adjust the embedded network to changing application requirements or environmental parameters. The last group are users which are not involved in the creation and maintenance of the embedded network, but will simply use the network for example by interacting with sensors or actuators. A crucial observation is that all these roles require different skills and a different level of knowledge about embedded networks. To keep management costs low, it is desirable that the manager and the installer role can be performed solely based on domain knowledge and do not require special training in embedded network technologies.

1.4.2.10 Different Workflows

Depending on the application field and the customer, automation solutions are developed using different workflows. One possibility is the offline modeling of the whole automation solution prior to the installation (or even acquisition) of hardware devices. This workflow is nowadays commonly used for the development of large scale automation systems. With the availability of smart devices on the field level, a more dynamic development is possible. By storing configuration and hardware information directly on the device, devices will become self-descriptive. Self descriptive devices can be used to model systems bottom-up. Whenever a self descriptive device is added to an existing automation solution, its capabilities and properties are queried by the system and fully automatically added to the system model. The developer can then integrate the new device into the existing applications. A vision for a development workflow without any prior modeling is a module based automation system. In this vision, an automation system, such as a manufacturing site, is created by installing modules for specific automation tasks, such as packaging, bottling or cleaning. These modules are automatically discovered and can be integrated using an iterative workflow in which the modules are added one after the other to the automation system. Of course, all possible variations between these two development workflows are possible, too. The development process used for building next generation automation systems should be flexible enough to support these variants.

1.5 Outline

This work is structured as follows. We will first present the core design concepts of the ϵSOA platform (also published in [130] and [129]) and provide a short overview of the application development workflow at the beginning of Chapter 2. In the remaining part of Chapter 2, we will present the ϵSOA system model in more detail, derive a service and instance lifecycle model and describe the service based application composition workflow. Finally, we will present the optimization techniques used in the ϵSOA platform at the end of Chapter 2 (parts of this work have been published in [132]).

After the presentation of the design principles and development workflow in Chapter 2, Chapter 3 focuses on the ϵSOA Runtime Environment, which provides the corresponding service execution environment on the nodes in the embedded network. We will present the overall architecture, the used description and encoding formats and present the communication layer (published in [131]), which is a crucial part in a distributed control network. We will describe how efficient message parsers can be implemented on resource constrained nodes (some of the used code generation techniques have been presented in [10]) and describe lightweight interfaces for the management of nodes and services and the information exchange between different layers of the communication stack (also called cross-layer information exchange). Furthermore, we will present how new services can be installed at runtime on nodes in the embedded network. This mechanism can be used to perform in-field updates and support end-user adaptations for off-the-shelf components.

Chapter 4 describes advanced features provided by the ϵSOA platform, which are based on the basic mechanisms introduced in Chapter 3. We will present failure compensation mechanisms for distributed control applications, a service migration scheme that allows the live migration of stateful services (published in [142]) and a bridge component that acts as mediator between services from the embedded domain and services from the IT domain (published in [141] and [11]).

In Chapter 5 we will provide a short overview of the development tools included in the ϵSOA platform, which implement the aforementioned concepts and support an easy and fast development of embedded networks. We developed demonstrators, which showcase the functionality of the ϵSOA platform. These demonstrators are presented in Chapter 6. Finally, a summary and an outline of future and ongoing work is given in Chapter 7.

CHAPTER 2

The ϵSOA Platform

The challenges for the application development in embedded networks summarized in the previous chapter cannot be solved by a single technological concept, but require a holistic solution that comprises a well-thought system architecture, an efficient runtime environment, a suitable development paradigm, and corresponding tool support. The ϵSOA platform, which we will present in this work, aims at providing all these features. The key design principles used in the ϵSOA platform are explained in the next section, followed by a more detailed description of the individual building blocks used to realize these design principles.

2.1 Design Principles

The variety of application fields and the different hardware platforms with unique capabilities and constraints make it very hard to build a general all-purpose middleware that can be executed efficiently and with little overhead on any system. Instead we aimed at designing a development platform that allows a user to specify what applications he wants to execute in the embedded network on an abstraction layer that safely hides the details of the underlying hardware. This abstraction is provided by employing the concepts of a Service Oriented Architecture (SOA). Based on this high-level specification, the ϵSOA platform will fully automatically create all code, including the runtime environment, that has to be installed at the nodes in order to run the applications. This transformation is performed using model driven development techniques, which allow creating highly efficient systems with minimal overhead. A primary design goal for the runtime environment was to efficiently support the distribution and processing of periodic a priori known data transfers, as these represent the vast majority of network traffic encountered in a control oriented embedded networks. An execution model that supports this kind of data transmissions and that can be well combined with a SOA is a data stream oriented processing paradigm. These three design principles are presented in more detail in the following sections.

2.1.1 SOA

The term Service Oriented Architecture (SOA) has become quite popular in the IT domain. A closer look at the big number of systems that are - or at least claim to be - based on the ideas of Service Oriented Architectures reveals a lot of differences w.r.t. the definition of SOA. When talking about SOA in this work, we are referring to a system architecture that follows the definition provided by the Organization for the Advancement of Structured Information Standards (OASIS). The definition is:

> *"Service Oriented Architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains."*[109]

As mentioned in the definition, SOA is not a technology for implementing systems but an architectural design principle. The focus of SOA is functionality. Functionality is bundled in services, which are described by a well defined interface specification that contains all information required for interacting with the service. Individual services can be combined to create applications. This process is called service composition. An important property of service compositions is a loose coupling between the individual services. Unlike compile-time or runtime binding, which require changes on the client and the server side if a service's interface changes, loose coupling gives service owners the flexibility to change the implementation and the interface of services independently of the other services used in a composition (of course with some limitations, because the new interface still has to provide the functionality required by the other services).

The architecture of the ϵSOA platform is based on the principles of SOA mapped to the domain of embedded networks. The capabilities encapsulated by services in the ϵSOA platform can be functionality provided by hardware devices, such as sensors and actuators, or data processing functionality. If a distinction between these two types is necessary, we will call the former kind of services "hardware services" and the latter kind "software services". Service compositions in the ϵSOA platform will typically involve both, hardware services (which provide access to sensor and actuator devices) and software services (which contain the control logic or provide storage, logging or monitoring functionality). However this is not mandatory and applications can be implemented by using solely hardware services, or software services respectively. We will use the term application synonymously for service composition. Note that applications do not have exclusive access to services, instead a service may be used in multiple applications simultaneously and a single embedded network can execute multiple applications in parallel.

SOAs have several properties that help solving the requirements mentioned in the previous section. The clear separation between the interface and the implementation of a service allows dealing with the heterogeneity encountered in embedded networks. A service can be implemented differently on different systems, tailored to the characteristics of the underlying hardware and software platform. The second concept that makes SOA an appealing solution, especially for heterogeneous systems, is the loose coupling of services. By using a suitable service description language and composition paradigm, applications can be designed in a way that does not require the presence of a specific hard- or software service but instead requires the presence of some functionality. In different systems, this functionality can be provided by different services. This allows building applications that are reusable in different systems, instead of re-designing the application for every embedded network. Consider an application that requires periodic temperature measurements. In one system, this functionality might be provided by a dedicated temperature sensor, in another system by a sensor board that contains humidity, temperature and brightness sensors. The same mechanism can be used to integrate devices from different vendors in a single system. The only prerequisite is that these devices provide the required functionality.

Another benefit of SOAs is the inherent support for a distributed execution of applications. SOAs encourage the decomposition of applications into bundles of interoperating services. These services can be distributed throughout the network. This mechanism is especially interesting for resource constrained devices. The burden of executing applications can be distributed over multiple nodes, thus reducing bottleneck situations. This flexibility can be extended to support network dynamics, too. Nodes can be implemented as general purpose service hosts and are therefore capable of executing arbitrary software services (and hardware services if they possess the required sensor/actuator devices). A system design based on such a paradigm can react to network dynamics, such as the failure or addition of nodes, by reorganizing the distribution of services. This allows building systems that can recover from node failures and are easily extensible. The envisioned goal is the following: in order to provide more processing power or to support a new application field, the user simply adds the required hardware resources to the embedded network. The network should then reconfigure itself to match the requirements of the new application field and to exploit the additional resources. As we will show in this work, SOA is a well suited architectural paradigm for building "smart" embedded networks that adapt their behavior to the application requirements and the capabilities of the underlying hardware.

SOAs are also beneficial w.r.t. the integration of IT systems and embedded networks. In the IT domain, the SOA concept - implemented with Web service technologies - is well established and used in many business systems. This eases the integration between the domain of embedded networks and the IT domain. Using the same architectural concept in both domains is essential for a seamless integration. An application developer should be able to use services from both domains and compose them to applications spanning embedded and IT services. Of course there are non-functional differences between services in both domains that have to be taken into account. The resource limitations in embedded networks require very resource efficient solutions and some features such as reliable message transfer or encryption support may not be offered by services in the embedded domain. Nevertheless, the fundamental concepts in both domains are the same and Web service knowhow is sufficient for integrating embedded services into IT systems. This similarity has another benefit. If embedded networks use a solution that is based on Web service technologies, or a solution that can be mapped easily to WS technologies, one can benefit from the vast investments done in the domain of Web services for training people and developing toolsets. This can dramatically reduce development costs compared to nowadays automation solutions, which are often based on specialized protocols that require highly trained experts and custom made toolsets during development.

2.1.2 Data Stream Oriented Processing Model

A second important building block besides the system architecture is the data processing model. As mentioned in the previous section, it has to efficiently support the distributed execution of applications on resource constrained devices. A processing model that satisfies this requirement is a data stream oriented processing model. In a stream oriented processing model, applications are composed of components that possess a well defined interface comprising inputs for receiving data and outputs for sending data. These components are connected by data streams. A data stream is a continuous flow of events/messages that connects an output of a component with the input(s) of one or more components. A component can be seen as a data stream operator that consumes data streams at its inputs and produces data streams at its outputs. An important characteristic of data stream processing systems is that operators/components are purely data driven. An operators output is determined solely by its configuration and the data it receives at its inputs. The source of a data stream is irrelevant for the operator. Analogously, the output streams produced by an operator are not influenced by the sinks that consume this stream.

A special characteristic of the communication patterns encountered control oriented embedded networks is longevity and predictability. Applications installed in control oriented embedded networks will typically be executed for days, months or even years. This makes the message flow occurring in the embedded network predictable. Of course there are also applications that are event driven, e.g., a fire detector that will send a signal only once a year - if at all. Nevertheless, the interaction between the fire sensor and an alarm bell is still predictable. It is known beforehand that if an event occurs, it will be send to the alarm bell. The fire detector does not select a destination ad-hoc. Because the communication is predictable, it is possible to optimize the message flow in the system. It is possible to exploit synergies between different applications that consume the same data and it is possible to route the data streams in a way that avoids network congestions and bottleneck situations. In the database community, the special term Data Stream Management System (DSMS) has been coined for management infrastructures that optimize the message flow in data stream based systems. The ϵSOA platform uses optimization techniques that are comparable to the optimizations performed by these DSMS. We will present DSMS and the used optimization techniques in more detail in Section 2.6.

A processing model that is based on independent components which communicate by exchanging messages is often also called an actor-oriented design. Actor-oriented models are well known in the literature for quite some time. An initial definition was given by Carl Hewitt[52], which was further refined in the work of Gul Agha[5] and others. A good introduction to actor-oriented system design and references to many related projects can be found in Lee et al.[86]. An actor-oriented design consists of actors with a well defined interface. The interface of an actor contains ports, which are used for communication with other actors, and parameters, which contain configuration information. The ports of actors can be connected via channels, which pass data between the connected ports. Like in the data stream processing model, an actor does not know to which endpoint the channel is connected to. The actor-oriented design does not imply any semantics for these components. Instead the semantics are provided by a *model of computation*, which is largely orthogonal to the definition of actors, parameters, ports and channels. The model of computation specifies rules that determine when an actor executes its internal logic, updates its state and communicates with other actors. The separation of actors and the model of computation allows building highly reusable actors that interact using the model of computation that fits best for a particular application field. An interesting feature of actor-oriented designs is the possibility to automatically generate hard- and software systems out of the actor model, see Lee et al.[86] for more details.

There is a close relationship between the Data Stream Processing Model and the Actor Oriented Programming Model. Both rely on distributed components that communicate by exchanging messages. Both enforce a strict separation between the implementation of components and the composition of components into applications. From an actor-oriented point of view, the data stream processing model can be seen as an actor oriented system using a dataflow/discrete event processing model. The focus of our work is closer to the tasks performed by DSMS, i.e., the optimization of the communication and data routing. We will therefore use the term data stream processing model and not actor oriented programming model throughout this work. Nevertheless, the concepts used in actor-oriented system designs fit seamlessly into the ϵSOA platform. Especially the possibility to create not only software but also hardware implementations could be an interesting feature for embedded networks and a possible direction for future research.

The stream oriented processing model fits well into a service oriented system architecture. Operators can be seen as services that offer operations that produce, consume or process data streams. The data driven execution model enforces the design of loosely coupled components. If the behavior of components only depends on the received data and not the source of this data, the source can be exchanged freely as long as the corresponding service provides a similar interface. The stream oriented processing model also provides a good way to deal with heterogeneous systems spanning IT systems and embedded networks. A common problem encountered in these systems are different addressing formats. The URLs used in the IT domain for addressing services can easily have a size of dozens of bytes. This is too resource consuming for embedded networks regarding the resulting message sizes and the overhead required for processing these strings. A benefit of a data stream oriented design is that the routing of data streams is done in the middleware and therefore transparent for an operator/component. This can be exploited during the installation of applications and the configuration of the data stream routing. If a user connects a service in the embedded network (identified by a numerical address) with a Web service (identified by an URL), the middleware can fully automatically generate an address mapping that translates the URL to a numeric address and vice versa. The embedded network is then configured to send messages directed at this numerical address to a bridge node. This bridge converts the address back to the original URL and submits the message to the corresponding Web service. This mapping can be done transparently for the end-user and the involved services. The end-user may still use URLs to identify services during application composition. However the middleware can decide to convert these URLs to other - more efficient - addressing schemes at runtime in order to improve the performance.

2.1.3 Model Driven Development Approach

The third key piece of the ϵSOA platform is a model driven development approach. Model Driven Software Development (MDSD), also called Model Driven Engineering (MDE), is a software development methodology that aims at reducing the gap between a problem domain and the software implementation domain. MDEs are based on models that describe a complex system at various levels of abstraction. Ideally, these models can be transformed directly into executable software using model transformation and code generation techniques. An overview of research in this area and the history of MDE is given by France and Rumpe[41].

Probably the most prominent example of MDE is the Model Driven Architecture (MDA)[82] specified by the Object Management Group (OMG[1]). The MDA advocates the creation of systems based on three models: a computation independent model (CIM), a platform independent model (PIM) and a platform specific model (PSM). The CIM captures the environment of the modeled system and the required features. The PIM contains all features of the system that will (most likely) not change from one platform to another. In this case, a platform is a framework such as CORBA or J2EE or some proprietary solution. Platform specific details are integrated into the system using the PSM. A typical development workflow based on MDA uses two transformation processes, a model to model mapping between the PIM and the PSM and an automatic source code generation based on the PSM.

A challenge encountered during application development for embedded networks is that the execution environment for an application is a dynamic, heterogeneous distributed system. The ϵSOA platform uses a MDE approach that is tailored for

[1]`http://www.omg.org/`

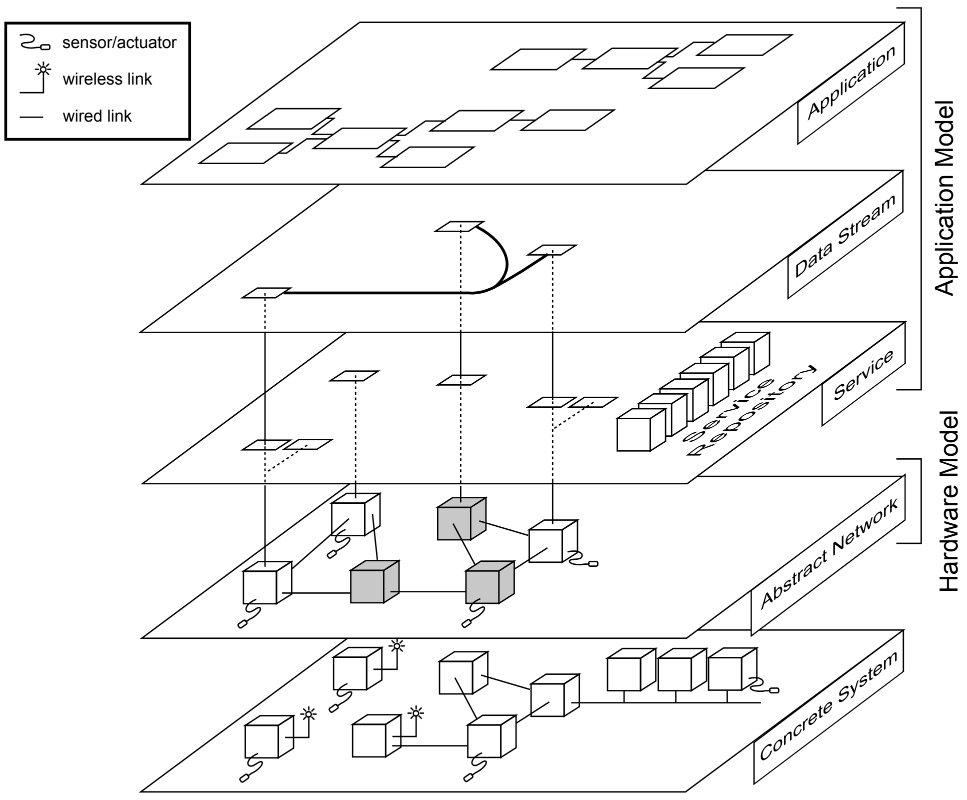


Figure 2.1: Abstraction Layers used in the ϵSOA Platform

application development in such environments. It uses an automated code generation to support different hardware platforms. The code generation is not only used to generate the applications running in the embedded network, but also to tailor the middleware executed on each node in the embedded network. The generated middleware for a specific node will only contain those features that are required by the services running on this node. This allows building a scalable middleware that supports both, simple nodes that only require a limited subset of features and complex nodes that require the full functionality of the middleware.

The system model used in our approach is divided into an Application Model and a Hardware Model. The combination of both models allows optimizing the execution of applications based on the characteristics of the underlying hardware and the requirements of the different applications. The ϵSOA platform uses four abstraction layers to model an embedded network, which are shown in Figure 2.1.

The *Abstract Network Layer* provides an abstract view of the available hardware and the communication characteristics in a concrete system. It possesses two elements, nodes and links. Nodes represent computing devices in the embedded network, such as microcontrollers and PCs. Nodes are annotated with information about the hardware platform, the operating system running on the node and its hardware characteristics, such as the available free memory. Depending on the hardware platform and the operating system, nodes can be re-programmable at runtime (indicated by the gray boxes) or offer only a fixed set of services (indicated by the white boxes). Each node contains a list of all attached sensor and actuator devices and their characteristics. The communication channels present in the embedded networks are represented by links between the nodes in the abstract network layer. These links represent a single-hop communication between nodes. If nodes are using broadcast media such as wireless connections, a link to every reachable node is added to the abstract network layer. The ϵSOA platform supports a variety of network protocols for the communication between nodes, it is even possible to use multiple different protocols in a single embedded network. The links are annotated with the characteristics of the network protocol used for communication. One source of characteristics are protocol specific features, such as the support for reliable message transfer, support for multicast and broadcast communication, or support for certain Quality of Service parameters, such as latency or bandwidth guarantees. The other source are characteristics of the underlying communication technology, e.g., the available bandwidth, the latency, etc. The Abstract Network Layer provides a simplified view of a concrete system. Often sensor and actuator devices are not attached directly to a node but connected via a bus system. The Abstract Network Layer contains only nodes and links that are interesting from an optimization/configuration point of view. If the communication between the sensor/actuator and the node is fixed, the sensor/actuator will be treated as if it were attached directly to the node. This situation is depicted in the lower right part of Figure 2.1. The bus system connecting a sensor/actuator device to the rightmost node in the embedded network is not visible in the Abstract Network Layer.

The *Service Layer* provides an overview about all services and service instances present in an embedded network. These services can be both, hardware services that provide access to a sensor or an actuator device, and software services that encapsulate a certain piece of application logic. The service layer also contains a service repository. The service repository stores software services, which can be installed on-demand on the nodes in the network in order to provide new functionality or to adapt the applications running in the network to new application fields.

The *Data Stream* layer models the communication between components in the embedded network. It contains an overview of all data streams running in the systems. Each data stream is originated by the output of a service instance and is consumed by one or more inputs of other service instances. Allowing multiple sinks for streams allows reducing the network traffic if a service is used in multiple applications, e.g., a sensor service. This is especially useful if the underlying network protocol does not support multicast communication, or if streams have to be transported across multiple subnets with different network protocols. The detection of re-usable data streams and high-level routing decisions are done based on this layer.

The upmost layer is the *Application Layer* that comprises all applications, i.e., service compositions, currently executed in the embedded network. It contains all information that is required during the installation of new applications or the reconfiguration or removal of already installed applications. For each application it stores the service instances that are used by the application and the data streams that have to be established for the communication between these instances. Additionally, the application layer stores the non-functional requirements for each application and the contained instances and data streams. These non-functional requirements specify, amongst others, the required amount of storage and CPU power and which hardware devices are needed for the execution of a service. Non-functional requirements for data streams typically comprise QoS requirements, e.g., concerning latency or bandwidth, or define features that have to be provided by the communication protocol used, such as reliable message transfer, encryption, etc. The Application Layer is also used to specify error-handling and failure recovery strategies for individual applications.

The model driven development approach is based on two models: the *Hardware Model*, which is based on the Abstract Network Layer, and the *Application Model*, which comprises the Application Layer, the Data Stream Layer, and the Service Layer. This design enforces a clear separation of the application model, which comprises a definition of the high-level communication paths and the requirements imposed by the applications, from the Hardware Model, which comprises the hardware characteristics and the capabilities and features offered by the underlying hardware. Both models are combined to a *system model*. During this process optimizations can be performed, which we will present in the following sections. This combined model is used to automatically generate code for the targeted hardware platform and to automatically configure the middleware and the executed services according to application requirements and network characteristics. The optimizations during the model transformation and the tailored code generation is essential for minimizing overheads and achieving as compact and as efficient code as possible and to support heterogeneous networks containing very lightweight microprocessors with constrained resources.

A special characteristic of embedded networks is that in many application fields, embedded networks cannot be preconfigured and simply deployed in a given environment. Instead they have to be (re-)programmed in-field, e.g., to allow the addition and removal of new applications or to support the creation of embedded networks out of standardized components. Consider a home/building automation scenario. The embedded network in such a scenario is created by connecting off-the-shelf components such as switches, relays and sensors and more complex devices such as heaters or air conditions. The final application model, i.e., the applications that should be executed in this embedded network, is specified by the owner of a building, not the developers of the individual components. These only provide specific parts of the Application Model, the final assembly is performed by the end-user himself. The ϵSOA platform supports such scenarios through a system design and corresponding development tools that allow different user groups to deliver parts of the application model and a service composition paradigm that allows even untrained users to compose an application model out of these individual parts.

In classical software engineering, MDE is often only used to create the software itself - configurations and adaptations are performed manually for each installation. This is not sufficient for embedded networks. The network structure can change over time, due to node failures, the addition of new nodes or changing environmental boundary conditions. The ϵSOA platform continuously monitors these parameters and keeps the Hardware Model up-to-date. If a persistent change is detected, the execution of applications is adapted to reflect the new boundary conditions. Possible adaptations are the reconfiguration of nodes and services and the relocation of services between nodes to optimize the transmission of data streams. The same procedure is applied if new applications are added to the embedded network or existing applications are modified. The Application Model is changed to reflect the new situation and the system automatically reoptimizes the embedded network. This reoptimization may also lead to the reconfiguration of running instances if this improves the overall performance of lifetime of the embedded network.

2.1.4 Runtime Environment

Each node contains a runtime environment that provides an execution environment for services and administrative interfaces for the management and the configuration of services. We will only present a short overview of the used concepts here. All concepts are explained in more detail in the following parts of this work. The data stream oriented communication is provided by a *Stream Router* on each node. The data produced by services is submitted to the Stream Router, which in turn dispatches the data to services on the same node or services on remote nodes. The dispatching is done based on a data stream routing table maintained in each Stream Router. It is possible to specify multiple destinations for a stream in this table. This can be used to split a data stream and supply multiple services with the data

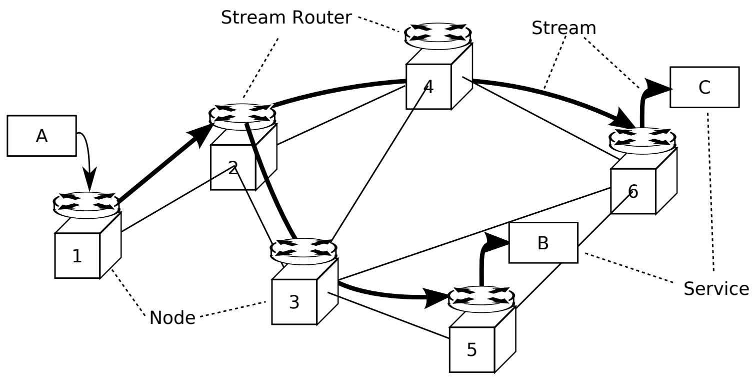


Figure 2.2: Example Application

produced by the stream source. An example application and the used data streams are shown in Figure 2.2. The data produced by Service A on node 1 is dispatched to node 2. The Stream Router on node 2 splits the data streams into two copies, one targeted at node 6, and the other at node 5. At the destination nodes, the data is submitted to Service C and B respectively. Note that Stream Routers specify the high level routing of data on the application level. The routing of individual packets in the network - possibly using multiple links - is performed by an underlying network protocol.

2.1.5 Summary

The three concepts presented in this section, are the central building blocks of the ϵSOA platform. The combination of SOA and model driven design principles are the key to an efficient execution of applications on heterogeneous networks with resource constrained devices. The decomposition of applications into compositions of cooperating services allows distributing the processing workload over multiple nodes. The services communicate using a data stream oriented processing model. The dissemination and processing of these data streams can be optimized based on the information provided by the model driven development approach. This allows an automatic adaptation of the execution of applications based on application requirements and network characteristics. The optimizations are not limited to the installation phase of an embedded network. The ϵSOA platform continuously monitors the underlying network and keeps the system model up-to-date. Persistent changes in the network structure, such as node failures, can be compensated by reorganizing and reconfiguring the embedded network.

Based on these key concepts, we designed a development platform for embedded networks: the ϵSOA platform. We will describe the individual building blocks of this platform in the following chapter, starting with an overview of application development workflows that show the interaction between the individual components in more detail. We will also describe solutions for the requirements which are not covered by the above mentioned concepts: end-user programming and the support of multiple user groups.

2.2 Development Workflow

The development workflow leading to a running embedded network can differ from application field to application field. In some cases special hardware has to be developed, in other cases off-the-shelf components can be assembled to build the embedded network. The same holds for the applications. In some cases functionality can be provided by software already available in some repository, whereas other application fields require the development of customized solutions. The development workflow in the ϵSOA platform is structured as shown in Figure 2.3.

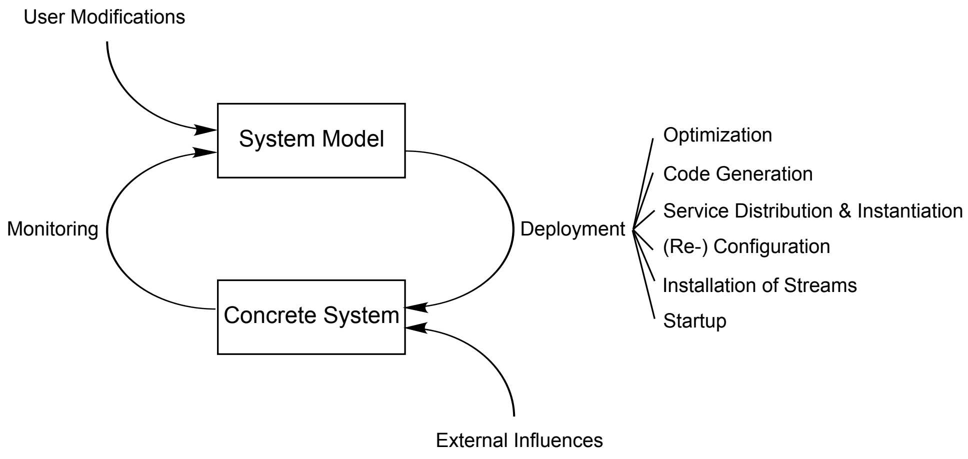


Figure 2.3: The ϵSOA Development Workflow

All application related information is maintained in the *System Model.* If a user wants to change the configuration of an application, or remove or add an application, these changes are first performed in the System Model. The same holds for the initial installation of the system. The desired applications are first added to the System Model. To propagate modifications to the *Concrete System*, the user can trigger the deployment process. The deployment is subdivided into several steps. In the first step, the application model and the hardware model are analyzed to determine an optimal execution strategy for the applications. If new services have to be installed in the network, the code generation is triggered to create a suitable implementation for the targeted node. After that new services are downloaded to the node and afterwards instantiated. In the fourth step, the configuration parameters of all instances and the configuration of the execution environment on the nodes is adjusted according to the requirements specified in the application. The fifth step is the installation of the data streams. The final step is to start the execution of applications.

An embedded network is not static. External influences can modify the network structure, e.g., the unexpected failure of nodes or the addition of new nodes by the user. Another source of variation are environmental changes that influence the

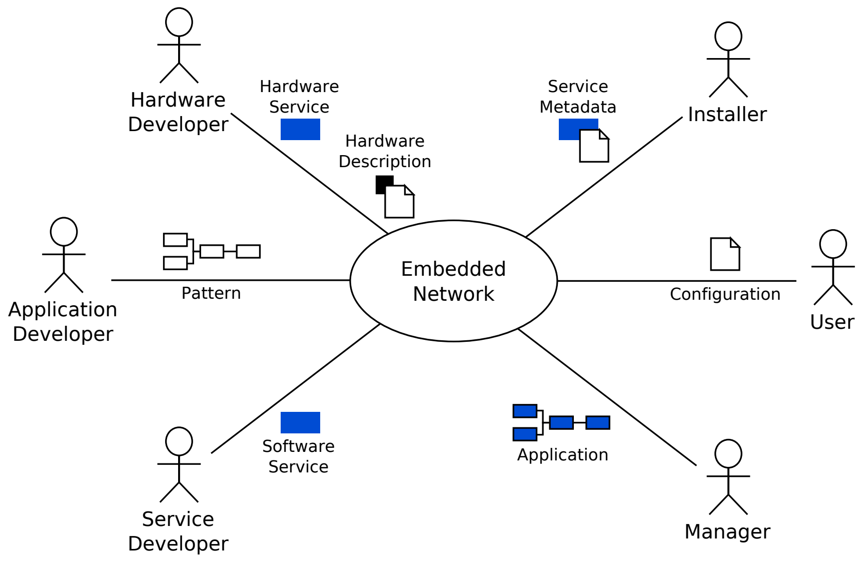


Figure 2.4: Actors Involved in the Development of Embedded Networks

characteristics of wireless links. In the ϵSOA platform, the nodes in the embedded network possess mechanisms to compensate transient changes and to provide a fast reaction to node or link failures. These mechanisms are based on a continuous monitoring of the network and its characteristics. If the change is persistent, the System Model is updated to reflect the new situation. By re-running the optimization algorithms, the system can check whether the change was substantial, i.e., check whether a different configuration will result in an increased performance. If this is the case, the deployment algorithm can be used to adapt the execution of applications to the new boundary conditions.

2.2.1 Actors Involved in the Development of Embedded Networks

The System Model is the junction point where hardware devices and software are assembled to an executable system. The different parts used in the System Model are widely independent and can be provided by different actors. We identified six different actors that contribute to the development of an embedded network. Figure 2.4 shows these actors.

The *Hardware Developer* is responsible for designing the hardware devices, including the node itself and sensor and actuator devices. The special characteristics of these devices are specified in a *Hardware Description* document, which is shipped along with the hardware. This document can either reside directly on the node or be kept in a central Web-accessible repository. In the latter case, the node only has to store a reference to an entry in this directory what is beneficial for nodes with very limited storage capabilities. The Hardware Developer is also responsible for developing hardware services that encapsulate the functionality provided by sensors and actuators and to create corresponding service descriptions.

Similar tasks are performed by the *Service Developer*. He realizes the control logic or other application logic by creating one or more software services and a corresponding service description. These descriptions can be stored together with the corresponding service in a service repository or can be pre-installed on nodes in the embedded network.

To increase the re-usability, many of the devices created by a Hardware Developer will be generic and usable for different tasks depending on their configuration. Consider for example a general rotary switch. It can be used to control various devices, e.g., to dim lights or to adjust the temperature in a room. The task of a device and its configuration are assigned by the *Installer* during the installation of a node in the embedded network. This information is added to the service description as additional meta-information.

The *Application Developer*, which is often also the Service Developer, creates application patterns. We will give a detailed description of patterns and their capabilities in Section 2.5 and only present a short overview here. An application pattern is an abstract definition of an application, i.e., a service composition. A pattern specifies a set of slots and connections between these slots. A slot is a placeholder for a service instance and specifies requirements that have to be fulfilled by this instance. The requirements comprise technical characteristics of service, such as the number of in- and outputs or the used data types, and semantic information specifying what kind of data is measured or consumed by a service.

A pattern can be instantiated in a given embedded network by assigning compatible services to each slot in the pattern. A service is compatible to a slot when it fulfills all requirements specified by the slot. The assignment of instances is performed by the *Manager*. When the pattern is completely filled, the application can be installed in the embedded network. This is done fully automatically by the ϵSOA platform by deriving a service composition from the pattern, installing the required data streams and configuring the involved services and nodes according to the requirements specified in the pattern. The Manager may also create applications by manually connecting services to applications. This approach should only be pursued by experienced users and is typically only beneficial if the created application should be installed only once (or used for testing and development purposes).

The last actor is the (End-) *User*, which will specify application specific configuration parameters and interact with the system through the provided interfaces.

A differentiation between these actors is useful, because the different actors require different knowledge about the embedded system. The Hardware and Software Developer must be able to program services using the interfaces provided by the ϵSOA platform and therefore need detailed knowledge about the platform and the programming of embedded devices. This is not true for the Application Developer. He must have some technical knowledge to understand how services can be combined and what functionality is provided by each service, but does not require detailed information about the service's implementation. Even less information is required by the three remaining actors. The Installer only needs basic domain knowledge to configure devices and to add the according meta-information to the service description[2]. The same holds for the Manager. He has to understand what the functional differences between services are, e.g., to select the most suitable control logic for a given application. The compatibility checks in the patterns ensure that only valid service compositions are created by the Manager. They can also be used to assist the Manager during the selection of services, e.g., by providing the Manager with a list of compatible services. The User requires the least knowledge of all actors. He may only adjust configuration parameters. This can be done purely on domain knowledge, assuming a suitable user interface is provided by the system.

The ϵSOA platform uses a development workflow that is split into multiple parts, each related to a specific actor. In Chapter 5 we will describe development tools that aid the individual actors in performing their tasks. The individual subparts of the development workflow can be executed independently of each other by the corresponding actor and can be re-arranged to support different, domain-specific workflows. We identified some typical application development workflows, which we will present in the following paragraphs. This list is not exhaustive and other workflows are also possible. Note that the different actors in these workflows do not necessarily have to be different persons. In many cases multiple tasks can be performed by the same person, e.g., the Service Developer and the Pattern Developer are often identical.

2.2.2 Workflow I: Assembly of Off-The-Shelf Components

In this workflow, an automation solution is created by combining standardized off-the-shelf components. In this case, the work of the Hardware Developer is strictly separated from the other actors because it is performed prior to the creation of the embedded network. The first step in this workflow is the installation and configuration of the hardware devices, what is performed by the Installer. After that, the nodes are started and the system will enter a discovery phase. In this phase, the Hardware Model is created by collecting the hardware descriptions of all nodes and by analyzing the network structure. Any pre-installed services on the nodes, e.g., hardware services, are detected in this phase and are used to populate the initial Service View. This view can be used to browse a pattern repository and retrieve application patterns that are compatible to the used hardware devices and a service repository to retrieve any missing software services required by these patterns. These steps are performed by the Manager and can be supported by tools that automatically filter the repositories based on the available devices in a concrete installation. When one or more patterns are filled, the Manager can trigger the installation of the pattern(s) according to the deployment workflow mentioned in the previous section.

[2]This can be done by selecting values from drop-down list or similar GUI components, and the Installer does not have to write a service description manually.

2.2.3 Workflow II: Off-The-Shelf Hardware, Custom Software

This workflow is an extension of Workflow I. The difference is that the software (or at least not all parts of the software) executed in the embedded network is not taken out of a repository. The ϵSOA platform supports two approaches for the implementation of custom software: a model-driven and a prototype driven development approach. Using the model-driven approach the user first models the desired service compositions and the interfaces of the individual services. This information can be used to create service stubs that have to be filled with application logic by the developer. Using the prototype driven approach, the user starts by implementing the application logic. Through annotations in the source code of the services, a service description is created automatically by the platform and added to the Service View. The developer can use these descriptions to quickly compose application prototypes out of the individual services. Both approaches may use hardware services provided by the embedded network. These are added to the Service View in a discovery phase similar to the one described in Workflow I.

2.2.4 Workflow III: Custom System

In this workflow, the whole embedded network is developed from scratch. The user starts with modeling the hard- and software used in the embedded network. In most cases, the hardware will be based on a generic microcontroller platform that is equipped with custom sensor and actuator devices. However it is also possible to create new types of nodes by specifying hardware characteristics such as available memory, communication interfaces, etc. Using this workflow, the developer also has to model the Hardware Services provided by the nodes because these cannot be discovered by inspecting a concrete system. The creation of software services can be performed following the two approaches mentioned in Workflow II. At any time, the developer may trigger the optimization module of the ϵSOA platform to check whether the system model is realizable or not. At the current stage, the optimizer can check fundamental characteristics, e.g., if enough memory is available on each node. We are currently working on extending these capabilities to provide Quality of Service related validations, too. One example would be to check whether a given response time requirement can be met or not. We will present these ideas in more detail in Section 7.

2.2.5 Workflow IV: Modification of Running Systems

The modification of a running system is no real workflow on its own but can be realized using any of the workflows mentioned above. The desired changes are added to the System Model. During deployment, the ϵSOA platform determines the difference between the running system and the modeled system and derives a series of actions that have to be performed on the live system to achieve the modeled situation. These tasks can either be basic operations, or operations that are especially designed for modifying a running system. We currently support 10 basic operations:

the installation/removal of services, the creation/deletion of service instances, the installation/removal of applications, the installation/removal/modification of data streams and the modification of configuration values. Up to now, we support one operation targeting a running system: the live migration (i.e. the state preserving relocation) of a service instance from one node to another, see Section 4.2 for a detailed description.

2.2.6 Summary

The development workflow used in the ϵSOA platform is based on a cyclic interaction between the System Model, which specifies the desired functionality of the system, and the Concrete System in a given installation. Changes in the System Model are deployed to the Concrete System using a deployment process that includes an optimization step for an automatic tuning of the data processing in the embedded network. Changes in the Concrete System are detected via monitoring and propagated back to the System Model. This basic cycle can be used to support different development workflows, ranging from specification first workflows to interactive workflows, in which an embedded network is created incrementally by adding more and more devices. The information contained in the System Model may be provided by different user groups involved in the development of embedded networks. The different user groups reflect the different classes of users - with different knowledge and programming skills - that interact with an embedded network.

2.3 System Model

We already gave a short description of the System Model used in the ϵSOA platform. It consists of a Hardware Model and an Application Model. The Hardware Model captures the structure of the embedded network and the characteristics of the used hardware. The Application Model is subdivided into three parts. The *Service Model* contains an overview of all services and service instances installed in an embedded network. The *Application Model* provides an overview of the applications composed out of these instances, whereas the *Data Stream Model* specifies all data streams used to realize the communication between instances. In this section, we will present an overview of all models. We will thereby focus on the information model. In Chapter 3, we will present XML description languages for each model and give a comparison with Web service standards having a similar focus.

2.3.1 Hardware Model

The hardware model is used to maintain an abstract view of the hardware devices present in an embedded network. This information plays a key role during the optimization of the embedded network. Based on the characteristics and constraints modeled in the hardware model, an optimizer is capable of configuring the system in a way that ensures an efficient execution and at the same time ensures that all application requirements and hardware constraints are met. Besides its use during optimization, the hardware model is also needed for validation. Prior to the installation of a new embedded network, or the reprogramming of an existing network, the hardware model is used to check whether the intended applications can be executed in the network or not[3]. During validation, the system can check the following requirements based on the hardware model:

Hardware Constraints Some services require specific hardware to be executed, e.g., sensor devices. During validation it is checked that this hardware is present on the node at which the service should be installed.

Resource Constraints Every service requires a certain amount of resources, e.g., ROM space for keeping the service code or RAM for storing its variables. This information is used during the planning and deployment process to to check whether the resources on a node are sufficient for executing all intended services or not.

Application Requirements Applications may specify non-functional requirements, e.g., an application may require a reliable message transfer. During validation, it will be checked if the requirements can be fulfilled by the embedded network or not, e.g., because a node does not support reliable network protocols.

[3]The optimizer will always produce a valid system. The validation is important because users may override the optimizations done by the system. By running the validation, it can be checked that the user defined system is still executable on the given hardware.

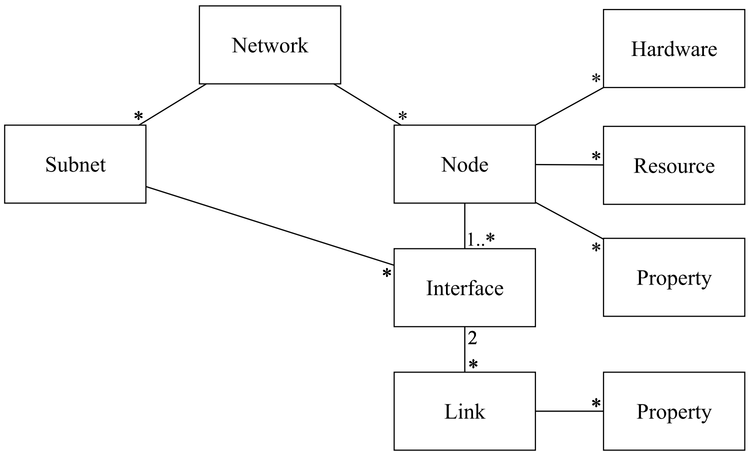


Figure 2.5: The ϵSOA Hardware Model

The level of detail used in the hardware model has to be chosen carefully. On the one hand, the hardware model should provide enough information to perform the optimizations and validations mentioned above. On the other hand, the hardware model should provide a high abstraction layer that allows optimizing applications across heterogeneous infrastructures, i.e., it should hide all unnecessary complexity to ease the optimization and provide an intuitive overview of the system for a user.

The hardware model presented in this section is suitable for building all demonstrators and examples shown in this work. Because the application fields for embedded networks are very diverse, the hardware model is not limited to these features, but can be extended to include additional information. This additional information is accessible during optimization and validation and can be included in the optimization process.

Figure 2.5 shows the hardware model used in the ϵSOA platform. An embedded network consists of *Nodes*. Each of these nodes has one or more communication *Interfaces*. The communication interfaces are grouped into *Subnets*. Nodes with more than one communication device will act as bridge between these subnets. Nodes that belong to the same subnet and have a single-hop communication channel are connected by a *Link*. Every link can be annotated with properties, e.g., the available bandwidth, reliability, latency, etc. Besides the communication interfaces, nodes also possess *Hardware* devices and *Resources*. The former are used to determine if a node can fulfill a certain hardware requirement of a service or not, e.g., if a node possesses a specific sensor device. The latter are used to check whether the resource capacities on a node are sufficient to execute all services assigned to this node. We currently use the amount of flash capacity and the amount of RAM as resources. A node can be annotated with *Properties*. These properties are used to add configuration, installation or administrative information to the node. Typical examples are inventory numbers, location information (e.g., the room number in

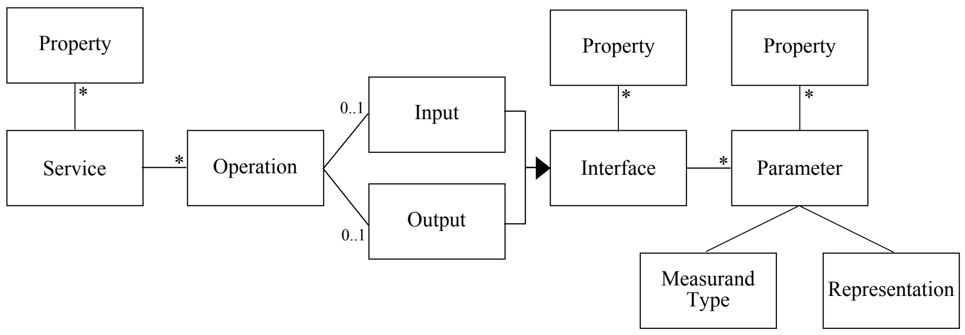


Figure 2.6: The ϵSOA Service Model

which the node is installed), etc. This information can be used to filter nodes during application composition, to search for specific nodes or to query nodes with a given property.

2.3.2 Service Model

The description of applications and services is a central aspect of every service oriented architecture. In the ϵSOA platform, the description of the Application Model is split into two parts. In this section, we will present the Service Model, which describes how individual services are modeled. The Service Model represents the Service View in Figure 2.1 on page 24. In the next section we will describe the Service Composition Model, which describes how services are composed to applications. The Service Composition Model represents the Data Stream and the Application View in Figure 2.1.

The ϵSOA service model is shown in Figure 2.6. A *Service* offers *Operations*. Operations can either consume data at an *Input*, produce data at an *Output*, or both consume and produce data. Throughout this work, we will synonymously use the term *port* to refer to In- and Outputs. *Interfaces*, i.e., In- and Outputs, possess a list of *Parameters*. Every Parameter has a *Representation*, which defines how the data is represented, e.g., as string, floating point value, etc. The *Measurand Type* specifies what type of data this parameter represents, e.g., a temperature value or a weight. Additionally, every Parameter can be annotated with a set of *Properties*, which refine the description of the data. Depending on the application scenario, varying properties may be of interest. Typical examples are the data unit, the precision, the possible data range, etc. *Properties* annotated at Ports define characteristics for the whole Port and typically cover aspects concerning data processing, e.g., the maximum frequency an output can deliver data with, the execution costs in terms of energy for an actuator, etc. *Properties* attached to Services contain installation specific metadata for a whole service, e.g., the location or configuration of a device.

An important field of the metadata description is the Measurand Type. It defines the meaning of a data value in terms of the application domain and will differ from

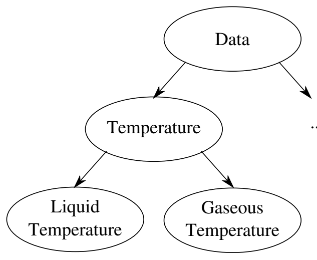


Figure 2.7: Example Taxonomy

scenario to scenario. We use a taxonomy for this purpose, i.e., the Measurand Type links to an object defined in a taxonomy for the given application domain. Figure 2.7 shows a very simple example taxonomy. The root of the taxonomy is the generic type *Data*. A possible specialization is *Temperature*, which is further refined into *Liquid Temperature* and *Gaseous Temperature*. Besides these simple meanings, the taxonomy allows to specify more complex elements, e.g., a fire warning service may define an output with a meaning "fire probability" that describes the possibility of forest fires depending on the air temperature and the humidity. This extensibility allows supporting arbitrary logic services which combine simple inputs to semantically richer outputs.

Service descriptions can be provided by different sources. Hardware services will typically be described by the manufacturer, logic services by the programmer of the service. During the physical installation of nodes, additional metadata can be entered by the Installer, e.g., the position and orientation of sensors and actuators or other data like inventory numbers. The third category of metadata is dynamic data, which represents the current state of nodes, such as energy resources or utilization, and is monitored during run-time. The ϵSOA middleware makes no assumptions about the presence of specific metadata fields, because these will greatly vary depending on the application scenario. Instead it offers generic filtering algorithms that support the extraction, monitoring, and configuration of subsets of nodes with given characteristics, e.g., all nodes which are in the same room, i.e., possess the same "room" and "floor" property.

2.3.3 Service Composition Model

An application in the ϵSOA platform is defined by: (1) a set of service instances and (2) a set of data streams that connect the ports, i.e., the in- and outputs of these instances. Because failure tolerance is an important criterion for many embedded networks, the ϵSOA platform allows the modeling of redundancy directly in the application model. During application development, the user may specify redundant instances for each instance used in an application. These instances are also stored in the application model and will be used to replace failed instances at runtime. Note that services that provide redundant functionality do not necessarily have to be exact replica of each other. It is sufficient if the replacements provide the same

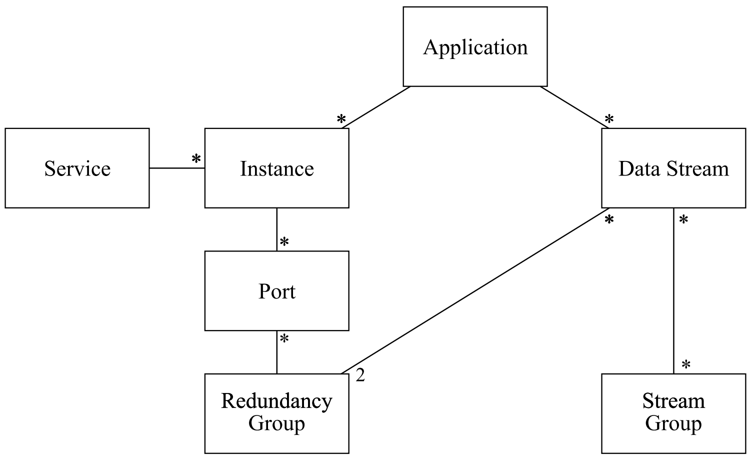


Figure 2.8: The ϵSOA Application Model

functionality. As a consequence, the service interfaces for redundant services may be different from each other. Consider for example a simple temperature sensor that provides temperature readings on output one, whereas a more sophisticated climate sensor provides temperature measurements on port two and humidity measurements on port one. Assume that we want to use the climate sensor as redundant backup for the temperature sensor.

In order to avoid ambiguities and to ease the selection of redundant instances at runtime, applications model redundancy based on ports. Ports that provide identical functionality are grouped into redundancy groups. A redundancy group may either contain inputs or outputs, never a mixture of both. In the example, port one of the temperature sensor and port two of the climate sensor would be put in a redundancy group. Data streams connect redundancy groups. If an instance fails, all redundancy groups can be checked to determine whether there is a redundant instance available that can replace the functionality of the failed instance. This mechanism is presented in more detail in Section 4.1.

The application model is also used to determine at which point in time an application has to be disabled because of failed instances. Disabling an application is an important decision in an embedded network. It allows saving resources by deactivating sensor devices that produce measurements that are not needed anymore. If an application depends on the presence of multiple actuators, a deactivation will not ensure that the application will not produce unwanted results if one of these actuators fails. The decision which failures are critical and should lead to the deactivation of an application is application specific. A simple case is the failure of a redundantly available instance. In this case the instance can be replaced and the application will continue to function as expected. But the opposite is not true in all cases. If a whole redundancy group fails, i.e., all instances from the group are unavailable, the corresponding data stream will fail too and can be removed from the system. However, the remaining parts of the application might still work as intended. One example is a lighting application that controls multiple lamps in a room at once. Even if one lamp fails (and there is no redundant lamp available), the lighting application should not be deinstalled. On the other hand the same lighting application can be safely disabled if the last lamp in the room fails. The application model uses stream groups to capture such dependencies. A stream group may contain one or more data streams of an application. The user can specify a lower limit on the number of availalbe streams in this group. If this threshold is underrun, the application will be disabled. Note that a data stream can be added to multiple stream groups. We were able to cover all practical scenarios encountered during the development of our demonstrators based on this mechanism. If a specific application field requires a more sophisticated ruleset, the data stream group can be easily extended to support other operations besides simple counting, too. The only prerequisite is that the ruleset has to be simple enough to be evaluated efficiently on a resource constrained node.

2.3.4 Properties

All three models presented in the previous sections use properties. Properties are $(name, value)$ pairs and can be used to add additional information to many elements of the system model. Every property is associated with a comparator function. This function is used whenever a comparison between property values is required, e.g., to check whether an instance is compatible with a slot or to search for services or nodes with specific properties. Example comparators are:

- **Equality**: used for enumerations other categorical data; requires that the value of properties is equal
- **Smaller/larger**: used for numerical values; requires that the value of a property is smaller/larger than the value of another property
- **Overlap**: used for properties that define data ranges; requires that the data range specified in both properties overlap
- **Contains**: used for properties that define data ranges; requires that the data range of a property is contained in the data range of another property

New comparator functions can be added if required and are automatically integrated into the development tools. Based on the available comparator functions, new properties can be added at any time to the system model, e.g. to add installation or runtime specific information to the system model.

In many application fields standardized information models for the description of systems have already been developed. Examples are the information model for automation systems in the industrial domain specified by the OPC Foundation[4] or

[4] `http://www.opcfoundation.org/`

the Industry Foundation Classes for building automation systems specified by the buildingSMART Alliance[5] or information specified with the Open Building Information Exchange (oBIX)[6], which is standardized by OASIS[110]. If a standardized information model is already available, the Measurand Type and Properties can also be defined based on a reference to this model.

2.3.5 Summary

In this section, we presented the hardware and the application model used in the ϵSOA platform. Both models define a basic structure but can be extended with application domain specific information. The model is stored in a XML based notation in the ϵSOA platform. In Section 3.4 we will present a service description language, in Section 3.5 a service composition language and in Section 3.3 a hardware description language. The XML formats were carefully designed to provide a compact representation of the information contained in the different models, what is a prerequisite for storing the models on resource constrained nodes. The motivation for storing (parts of) the system model on an embedded node is that an embedded network should be self-descriptive, i.e., a user should be able to retrieve all model information related to an embedded network by querying the nodes contained in this network. Based on this information, any user with a suitable toolset can manage and monitor the network without a prior synchronization with a central model repository.

[5]`http://www.buildingsmart.com/`
[6]`http://www.obix.org/`

2.4 Service and Instance Lifecycle

An important building block of service oriented systems is the service/instance lifecycle management. The lifecycle management in embedded networks has to meet several requirements. First the number and type of services installed at a node are not fixed but may change throughout the lifetime of the network, e.g., to adapt the network to new application fields or environmental changes. The service lifecycle has to support such reconfigurations by allowing a dynamic installation and instantiation of services. The second requirement is an efficient support of a stream based execution model. The processing of new data should create as little overhead as possible to allow an efficient execution on resource constrained nodes.

We will first analyze the service lifecycle used in nowadays Web service stacks and describe the limitations that prohibit its application in embedded networks. We will then present the ϵSOA lifecycle model that is tailored to the special characteristics imposed by embedded networks.

2.4.1 Web Service Lifecycle

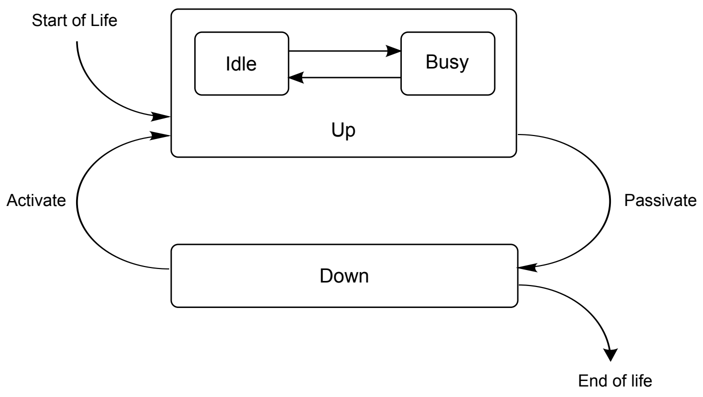


Figure 2.9: The Web Service Lifecycle

The W3C working group describes the service lifecycle and request processing for Web services in the Web Service Management: Service Life Cycle note[161]. The Web service lifecycle, as specified in the Web Service Management specification comprises two states: a service can either be “up”, i.e., it is ready to process requests, or “down”, i.e., it is not available and will not respond to requests, see Figure 2.9. Services that are “down” can be “activated” to make them ready for request processing or “passivated” to revert the state back from “up“ to “down” and stop request processing. Services that are “up” can either be “idle”, i.e., they are not processing request, or “busy”, i.e., they are currently processing one or more requests. The processing of requests is done by agents that take an incoming request, process it, and deliver the results (or an error message) to the requester.

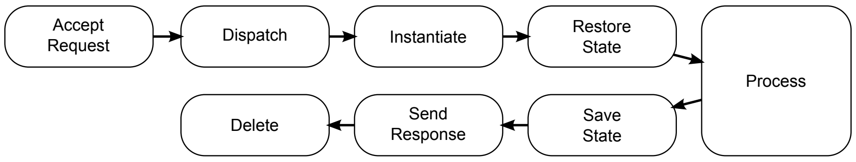


Figure 2.10: Web Service Request Processing Model

An important characteristic of the Web service lifecycle model is that the service model is inherently stateless. There is no notion of instances, which possess a persistent state between service invocations. When a stateful interaction with Web services is required, the state of this interaction has to be stored at an external location, such as a database. This state (and therefore implicitely a specific instance of a service) is referenced by the client either via a session identifier or a mechanism such as the Web Services Resource Framework (WSRF), see WS-Resource[111] and related standards. The invocation of a stateful Web service comprises the following steps, which are also shown in Figure 2.10:

1. Parsing of request and parameters
2. Dispatching of the request to the specific service
3. Instantiation of the service
4. Loading of instance state (based on the instance identifier)
5. Execution of service logic
6. Storing of instance state
7. Delivery of response
8. Deletion of instance

This lifecycle model is well suited for the use with stateless application protocols, such as HTTP[158]. Furthermore, it allows building servers that are capable of handling a large number of simultaneous interactions within a small amount of memory. Because all services are persisted between invocations, only the services that are currently being processed have to be stored in the main memory of the server. This is especially beneficial if there is a long time between service invocations targeting a stateful service. This is often the case for long-running business processes that may take days or weeks to complete but only perform a small number of Web service invocations during this period. The drawback of this approach is that each service invocation causes a considerable overhead for restoring and serializing the service state.

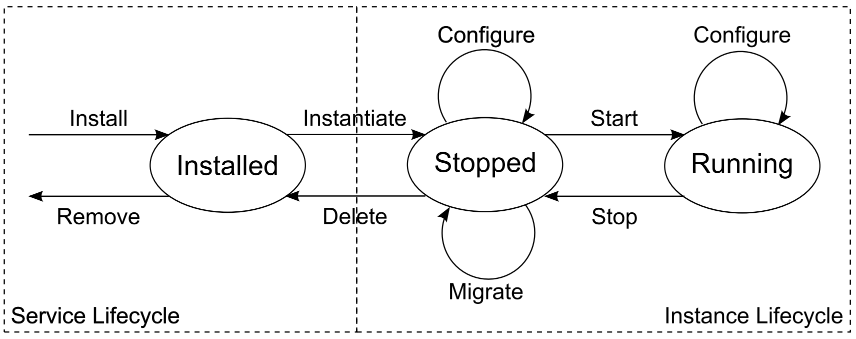


Figure 2.11: The ϵSOA Service/Instance Lifecycle

2.4.2 ϵSOA Lifecycle Model

The invocation pattern encountered in embedded networks is quite the opposite of the invocation pattern in the Web service domain. In the IT domain, a node offering Web services is faced with a multitude of users that access a large number of service instances, but each service instance is only accessed a few times. In an embedded network each node runs a few services, but these services are accessed very frequently, e.g., to process periodic measurements. The overhead for (de-)serializing service states between invocations is too large in this situation. The second difference is the lifetime of instances. Services involved in a control or automation task will be running for days, months or even years. During this time, service instances can be reconfigured, moved between nodes or even updated with a new implementation of the service. During these operations, the service instance has to be "stopped", i.e., the processing of data has to be halted until the operation is completed. A stopping or pausing of instances can also be triggered by the end-user, e.g., during the installation of new devices. The Web service lifecycle model does not support such operations because there is no notion of instances - the WS lifecycle focuses solely on services.

The ϵSOA lifecycle model comprises a *Service Lifecycle* and an *Instance Lifecycle* as shown in Figure 2.11. The Service Lifecycle comprises one single state, the state "installed". This state is reached when a service is installed at a node. Installed services can be instantiated to create service instances. Every service instance in the ϵSOA platform is stateful, i.e., will keep its internal state between invocations. Service instances are not persisted between invocations. The number of instances that can be executed by a node is therefore limited by the available memory. There is no overhead for (de-)serializing service states and frequent service invocations can be handled efficiently (several dozens of invocations per second are possible even on very lightweight microcontrollers). A single service may be instantiated multiple times, e.g., to create multiple instances of a logic service that can be used in different applications.

A service instance is initially in the "Stopped" state. In this state, configuration parameters can be changed, or a service migration to another node can be triggered. The migration process is explained in more detail in Section 4.2. In the stopped state, the instance will not receive or send any messages or execute periodic tasks. However it will keep its internal state. If a stopped instance is started it will enter the "Running" state. In the running state, the instance may interact with other instances by sending or receiving messages and it may execute periodic tasks. As in the stopped state, services can be (re-)configured in the running state. If a running service is stopped, it will enter the "Stopped" state again and the message handling and execution of periodic tasks is suspended. Stopped service instances can be deleted, which will destroy the internal state of the instance, free all used resources and remove the instance from the system. If there are no more service instances for a service, the service may be removed. If a service is removed, its binary code is removed from the node and the service is unavailable for further instantiation.

Depending on the hard- and software environment, not all nodes support every part of the lifecycle model. If the node uses an operating system that is not capable of dynamically loading code, such as TinyOS, it is impossible to install new services at runtime. In this case, the lifecycle is restricted to the instance lifecycle. The preinstalled services at a node can be instantiated, started, stopped and configured, however it is impossible to install or delete service instances. Such limitations can be specified in the hardware model of the ϵSOA platform. To circumvent this problem, the ϵSOA platform allows the installation of service libraries on nodes. A service library can be added during the initial deployment of the node and may contain an arbitrary number of services, only limited by the available storage on the node. Services from this library can be instanciated even if no dynamic loading of code is possible. Service libraries can greatly increase the flexibility of embedded network installations based on TinyOS and operating systems with similar limitations[7].

2.4.3 Summary

The ϵSOA service lifecycle has two main differences compared to the service lifecycle known from Web services: it explicitly models service instances and does not persist service instances between invocations. These changes were motivated by the special characteristics of embedded networks. Service invocations in embedded networks can occur with very high frequencies and the overhead for persisting the service state between these invocations is too large. The second reason is that the lifetime of service instances in embedded networks is very high (up to several years). Due to this reason, the ϵSOA lifecycle model also supports the reconfiguration and the migration of services. To the best of our knowledge, we are the first to propose a lifecycle model tailored for embedded networks.

[7]The Contiki based version of the ϵSOA runtime supports the dynamic loading of code. The library mechanism is offered as a convenience method if a migration to another operating system is not possible, e.g., due to missing driver support. The full flexibility and optimization potential is only achievable if the underlying operating system supports the dynamic loading of code - or if the used nodes have enough storage capacities to store all services used in the network.

2.5 Application Development

The manual specification of service compositions is a tedious and error prone task. To ease the composition of applications, the ϵSOA platform offers two development approaches: manual composition and pattern based composition. Both approaches are supported with corresponding development tools, which are presented in more detail in Chapter 5. In this section, we will present the underlying foundations in more detail. The final outcome of both approaches is a service composition based on the model presented in Section 2.3.3. Newly created or modified service compositions are added to the System Model and can be deployed to the embedded network using the development workflow explained in Section 2.2.

2.5.1 Manual Composition

Manual composition is a bottom-up service composition approach. The developer first specifies a set of services instances that should be used in the application. These instances can either be existing instances, i.e., instances already installed in the embedded network, or new instances based on services from the service repository. After that, the developer manually connects the outputs and inputs of these instances.

With corresponding development tools, this approach can be extended to support a progressive creation of service compositions. In this case, the developer starts with a specific service instance, e.g., a logic service. The development tool can automatically search for services with matching interfaces for each of the in- and outputs of this service. The developer can choose from this list and does not have to search the whole service repository manually.

The manual composition approach gives the developer full control about the resulting service composition. The developer must have fairly detailed domain knowledge to perform a manual service composition. The development tools can ensure a basic compatibility between services, i.e., can reject connections between in- and outputs using incompatible data types. If multiple services are available for a specific task, e.g., different implementations of the service logic requiring different sensor devices, the developer has to select a suitable service himself.

2.5.2 Pattern Based Composition

The requirement for end-user programming cannot be fulfilled with a manual service composition. Most end-users will neither have the required programming skills, nor the experience in developing control applications, required for manual composition. The *pattern based composition* is based on a clear separation between the application developer and the end-user (or more precisely the manager of a building according to the actors wie identified in Section 2.2.1).

This separation is achieved through the introduction of application patterns. An application pattern is an abstract, i.e., not directly executable, definition of a service

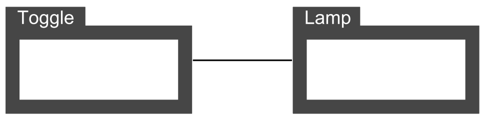


Figure 2.12: Lighting Pattern

composition. A pattern specifies a set of "slots" that can be filled with services in a given installtion. Each slot specifies an interface that has to be provided by the service, i.e., the number and type of in- and outputs offered by the service. The pattern additionally defines connections between the in- and outputs of slots. Based on these connections and an assignment of services to slots, the ϵSOA platform can automatically generate a service composition that can be installed in the embedded network.

Patterns are created by the application developer. The end-user/manager has to fill the slots of the pattern with suitable services from his concrete installation. With corresponding tool support, the assignment of services to slots can be performed by selecting from a list of compatible services for each slot. Figure 2.12 shows a simple example of an application pattern. It defines two slots, a "Toggle" slot and a "Lamp" slot. The "Toggle" slot can be filled with services that possess a single output that delivers on/off signals, e.g., toggle or rocker buttons. Services that are compatible to the "Lamp" slot have to possess an input that consumes on/off signals and no outputs, e.g., a service representing a conventional light bulb or a neon lamp. Furthermore, the pattern specifies that the output of the "Toggle" slot is connected to the input of the "Lamp" slot.

A slot in a pattern is defined using the service model introduced in Section 2.3.2. It is possible to define the number and kind of ports a compatible service has to possess and the number and data type of parameters consumed by these ports. Using the Measurand Type field, it is also possible to specify which kind of data a service has to provide, e.g., temperature measurements. These basic requirements can be further refined with properties. As mentioned in Section 2.3.2, the properties used in the ϵSOA platform are not fixed and can be tailored to the application field. A slot in a pattern can for example require a service that provides a temperature measurement with a resolution of at least tenth of degrees (with a numeric property "resolution" and the "larger" comparator) in a temperature range from -20°C to 40°C (with a range property "measurementRange" and the "contains" comparator).

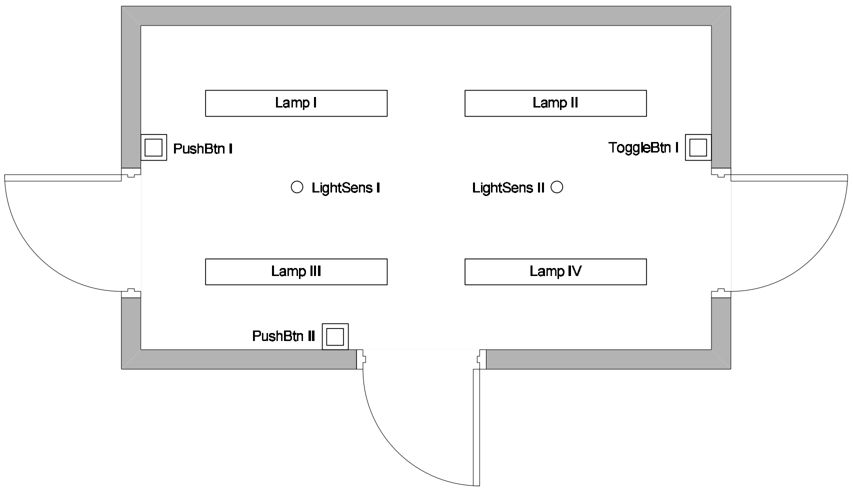
Application patterns have several benefits. The assignment of services to slots requires only basic domain knowledge and can be performed by an untrained user. Some assignments can be performed fully automatically, e.g., when only a single service qualifies for a slot in a pattern. For a home automation scenario, we envision the following scenario: a user buys a set of off-the-shelf automation devices and installs them in his home. After the installation of the hardware, a service composition tool automatically searches a repository for application patterns that use this hardware. The user selects one of these patterns. The development tool now automatically retrieves all software services required for filling empty slots in this pattern. The tool will also try to assign hardware services to the slots. If multiple services qualify for a slot, e.g., because multiple devices of the same kind are installed (e.g. multiple switches), the user has to select the appropriate one. When all slots are filled, a corresponding service composition is created and installed in the embedded network.

An interesting idea w.r.t. this vision is a community driven management of the pattern and service repository. An initial set of patterns could be supplied by hardware vendors or companies specialized in developing automation solutions (possibly with an according business model[8]). Further patterns and services may be added at any time, e.g., to support new hardware devices or cover new use cases. A community driven approach allows solving many of the scalability challenges imposed by the envisioned Internet of Things. With an increasing number of networked devices, the number of possible combinations of these devices will explode. It is questionable whether a small number of companies will be able to offer automation solutions for each of these combinations. With a community driven approach, it is possible to leverage the programming and domain knowledge provided by the users of these systems. This will hopefully lead to a situation where the number of developers will grow with the number of application fields, thus creating the required scalability. The recent development regarding the application development of smart phones shows that there is a considerable amount of programming knowledge available. Users and companies created and are still creating thousands of applications for modern smart phones. If we succeed in creating a comparable application development environment for automation applications, we will hopefully see a similar trend in this sector - especially because remote control of automation devices via smart phones will most likely be one of the first applications demanded by the end-user. We think that the combination of application patterns and SOAs is a development approach that is flexible enough to support different hardware devices but still simple enough to be developed in a community driven approach.

A second benefit of patterns is that patterns are re-usable. This can lead to a considerable reduction of development time, especially in large networks which contain many subnets with similar purpose. A good example is an office building, which contains many rooms (subnets) with similar sensors and actuators. A pattern has to be developed only once and can then be used to install applications in an arbitrary number of rooms. This installation does not have to be performed by the programmer but may also be performed by a third party, such as a facility management service.

Patterns also possess another interesting property: patterns externalize domain knowledge. The development of automation solutions requires not only the implementation of individual services but also the composition of these services to appli-

[8]One possibility would be to adopt a concept like the Apple App Store(`http://www.apple.com/iphone/apps-for-iphone/` for selling logic services and patterns on a Web based platform.



(a) Roomplan

Service	Inputs		Outputs	
	Nr	Type	Nr	Type
PushBtn I			1	signal
PushBtn II			1	signal
ToggleBtn I			1 2	signal on/off
LightSens I			1	brightness
LightSens II			1	brightness
Lamp I	1	on/off		
Lamp II	1	on/off		
Lamp III	1	on/off		
Lamp IV	1	on/off		

(b) Hardware Services

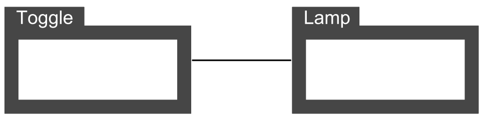
Figure 2.13: Building Automation Scenario

cations. Application patterns can be used to store and exchange common "patterns" encountered in these service compositions. Patterns therefore have a functionality in embedded networks that is comparable to the functionality of process modeling languages in IT and business environments.

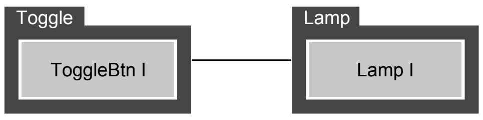
We will present the features and capabilities of application patterns in more detail in the following section using examples from a building automation scenario.

2.5.2.1 Building Automation Scenario

To illustrate the features and the application development workflow based on pattern based service composition, we will use an example from a building automation scenario. Figure 2.13(a) shows the scenario and the available hardware devices. The scenario is based on a room in a smart building that possesses three switches, two



(a) Pattern



(b) Service Composition

Figure 2.14: Basic Pattern for Direct Connection of Switch and Lamp

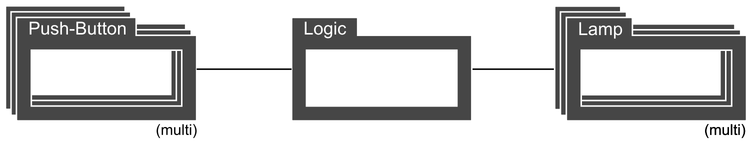
push buttons[9] (PushBtn I and PushBtn II) and one toggle button (ToggleBtn I), four lamps (Lamp I to Lamp IV), and two light sensors (LightSens I and LightSens II). Each of these devices is represented by a corresponding hardware service in the ϵSOA platform. Based on these devices, a lighting application is going to be installed by the end-user.

Table 2.13(b) shows a short summary of the service metadata of each device. The push buttons possess only one single output that is used to transmit a simple signal indicating the button was pressed. The toggle button possesses two outputs. The first output sends on/off signals whenever the state of the button changes, the second output sends a simple signal whenever the button is pressed. Both of these outputs are marked as optional, so the user can chose to either use the switch as a simple push button (by using the second output) or as a stateful on/off switch (by using the first output). The light sensors possess a single output that is used to distribute the current brightness to connected services. The lamps possess no outputs, as they represent actor devices, and one input that allows to turn the lamp on or off.

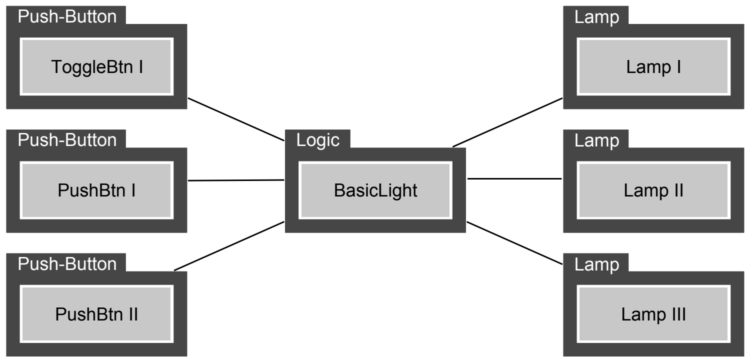
2.5.2.2 Application Patterns

Toggle Button Pattern Assume the user wants to start with a basic lighting application: he wants to connect the toggle button to Lamp I. A pattern that allows the creation of such an application is shown in Figure 2.14(a). It defines two slots, a “Toggle” slot and a “Lamp” slot. The Toggle slot can be filled with services that possess a single output that delivers on/off signals. Suitable services for such a slot are for example hardware services that represent toggle or rocker buttons. Services that are compatible to the Lamp slot have to possess an input that consumes on/off signals and no outputs, e.g., a service representing a conventional light bulb or a LED-lamp. Furthermore, the pattern specifies that the output of the “Toggle” slot is connected to the input of the “Lamp” slot. Given the running example, there is

[9]In this scenario, a push button denotes a switch that will always flip back in its original position when it is released. In contrast to this, toggle buttons have two states (on/off) and will alternate between these two states when the button is pressed.



(a) Pattern



(b) Service Composition

Figure 2.15: Advanced Pattern for Multiple Push-Buttons Connected to Multiple Lamps

only one compatible service for the Toggle slot, ToggleBtn I, and four compatible services for the Lamp slot, Lamps I to IV. Note that ToggleBtn I is only compatible to the slot, because its output “1” is marked as optional. If output “1” had been marked as required, the service would not match the slot in the pattern because it possesses two outputs and the slot requires a service with one output. Based on the filled pattern, a simple service chain will be created, which connects the output of the button with the input of the lamp. This chain is shown in Figure 2.14(b). In the figure, the service instances are annotated with the pattern slot they have been assigned to.

Combination Strategies To allow turning the lights on or off from every door of the room, the user decides to install a more advanced lighting application that also uses the push buttons available in his room. Additionally, he wants to control all lights simultaneously, and not only Light I. It would be a tedious task and in many cases infeasible to design patterns for every possible combination of instances, e.g., a pattern for two switches and one light, two switches and two lights, two switches and three lights, etc. To support building applications with an a priori unknown number of instances, slots in a pattern can be marked to support the assignment of multiple instances by specifying a *combination strategy*. The ϵSOA platform supports several combination strategies, which will be explained in the following paragraphs.

The most basic combination strategy is the *multi* strategy: it specifies that the outgoing data of all assigned instances is merged to one data stream and that incoming data is delivered to all assigned instances. A sample pattern using this combination strategy is shown in Figure 2.15(a). In the pattern, the “Lamp” and the “Push-Button” slots are annotated with the combination strategy *multi*. As a consequence, the output produced by the logic slot is distributed to all service instances assigned to the lamp slot. It is therefore possible to control an arbitrary amount of lamps simultaneously. Analogously, all data produced by instances assigned to the push-button slot is merged and submitted to the input of the logic service.

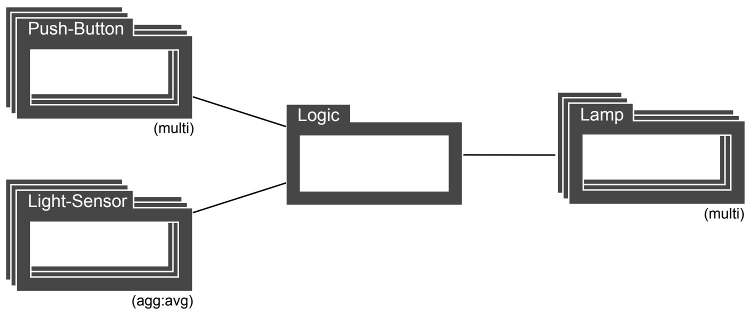
A possible service composition for the running example is shown in Figure 2.15(b). Both pushbuttons, PushBtn I and PushBtn II, have been assigned to the Push-Button slot. The toggle button ToggleBtn I can also be assigned to this slot because it possesses an optional output that delivers signals whenever the button is switched. The events produced by all these three buttons are sent to the same input of the BasicLight logic service. This service produces on/off signals which are submitted to all services assigned to the Lamp slot. In the example, the user assigned three lamps, Lamp I to III.

The multi combination strategy is typically used to control multiple actuators at once, such as the lamps in the example. In some cases, it can also be used to integrate multiple sensor devices, like the push-buttons in the example. This is only possible for sensors measuring data on an event basis. If the data is acquired periodically, one typically wants to have a some kind of integration between the data streams produced by the sensors, for example by calculating average values. A combination strategy that supports this kind of integration is the “agg” strategy explained in the following paragraph.

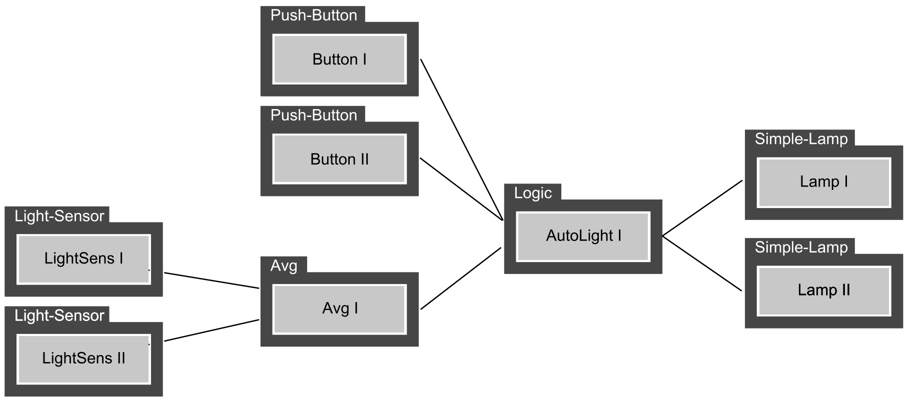
Aggregation of Data Streams Figure 2.16(a) shows a more advanced lighting application. The switches and lamps are used in the same way as in the last example. The pattern uses a different logic service that additionally consumes the data of one or more brightness sensors. The brightness information is used to automatically turn the lights on if it is too dark in the room[10]. The light sensors in the example are annotated with the “agg” combination strategy. The agg strategy specifies that all output data streams are sent to an aggregation service which combines all incoming streams to a single, aggregated output stream. This stream is then submitted to the input of the service instance(s) assigned to the connected slot(s). The aggregation service is automatically added during the installation of the application. The pattern developer can choose from different aggregation services, e.g., averaging, maximum, minimum, voting schemes such as majority consensus, etc. In our example, the pattern specifies that the measurements of the light sensors should be averaged.

A resulting service composition using the devices from our running example is shown in Figure 2.16(b). The upper part of the service composition is similar to the

[10]It would be a bad idea to turn the lights on if there is nobody in the room, so in a real world application such a mechanism should be combined with an activity/presence sensor that is capable of detecting whether a human is present in a room or not.



(a) Pattern



(b) Service Composition

Figure 2.16: Advanced Combination Strategies

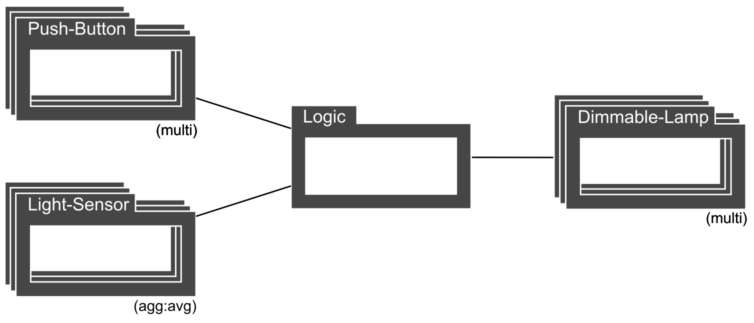


Figure 2.17: Advanced Combination Strategies in Patterns 2

service chain shown in the previous example. The data of all services assigned to the “Light-Sensor” slot is submitted to the aggregation service “Avg I”. This service creates a combined data stream by calculating the average of all input streams. The resulting stream is then submitted to the logic service “AutoLight I”.

Nested Patterns The above mentioned light control, which automatically turns on the lights based on the measured brightness, is only a first step in the direction of a “smart” room. It is quite difficult to adjust the threshold for turning on the lights, especially because this threshold may be different from user to user. A much better lighting application, which can be built out of the same components, is an application that automatically dims lights in a way that ensures the brightness always has a specific level. If it is daytime and the weather is bright, the lights are turned off. If its getting darker, the lights are turned on and are dimmed in a way that compensates the degrading brightness of the daylight. A possible pattern for such a lighting application is shown in Figure 2.17. Again, the lighting logic consumes the input from the buttons and the averaged input of the lighting sensors. However the pattern specifies dimmable lamps instead of the simple lamps used in the previous example as actors. The dimmable lamps accept a percentage value between 0 and 100% as input and will dim their brightness to the given level. To keep the example concise, we omitted a possibility for the user to adjust the desired brightness level. The lamps installed in the example scenario are not dimmable and therefore do not match with the specification in the pattern. On the other hand, the room contains a total of four lamps, so a possible way to mimic a dimmable lamp would be to selectively turn on or off a subset of the four lamps. This requires some additional control logic that takes the desired brightness percentage as input and turns on or off a subset of the connected lamps.

The ϵSOA platform allows using nested patterns for this purpose. A nested pattern is a pattern that itself possesses in- and/or outputs. When a nested pattern is installed in a system, it appears as a service that has the in- and outputs specified in the pattern. This service can be assigned to a slot just like any other service. Nested patterns can be used to reduce the complexity of large patterns by subdi-

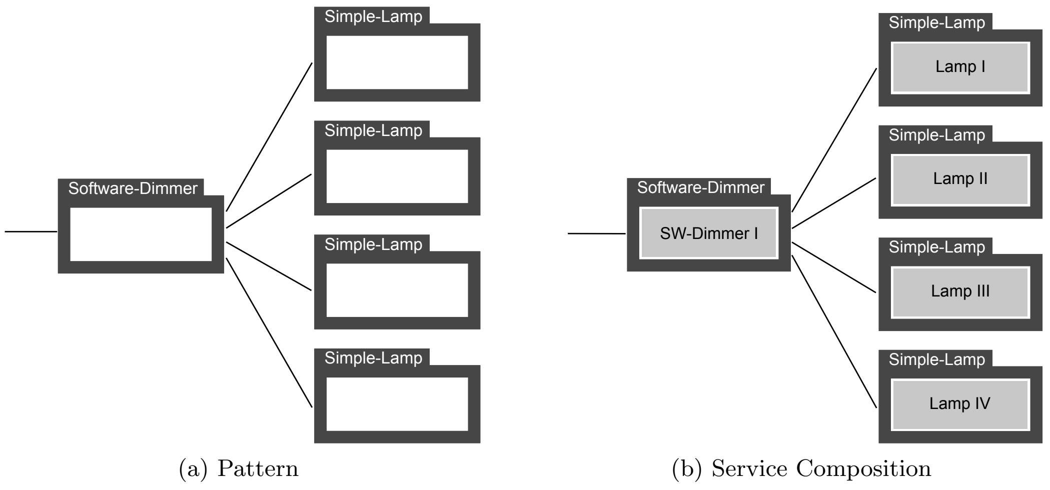


Figure 2.18: Nested Pattern for a Software Dimmer

viding the patterns into smaller subsets. Another application field is the extension of patterns with new functionality. As already mentioned, we would like to use a software dimmer to create dimmable lamps that can be used in the pattern shown in Figure 2.17. An example nested pattern for this task is shown in Figure 2.18(a). The pattern possesses a single input, denoted by the line on the left hand side, a slot for a logic service and four slots for lamp services. Note that we cannot use a "multi" combination strategy for the lamps, because we want to control each lamp individually and not turn on or off all lamps simultaneously. Given the services from our running example, the pattern can be filled with service instances as shown in Figure 2.18(b).

The completed pattern can now be used like a service with a single input and is compatible with the "dimmable-lamp" slot shown in Figure 2.17. A possible service chain using this nested pattern and the service instances from our running example is shown in Figure 2.19. Nested patterns are a modeling concept. During the installation of an application, all nested patterns are inserted into the application. In the example, the "Dimmable Lamp" slot of the pattern is replaced by the service chain created by the nested pattern, i.e., the "SW-Dimmer I" service and the connected lamps "Lamp I" to "Lamp IV". All data streams targeting the nested pattern are connected directly to the service instances assigned to the corresponding slots inside the pattern. Due to this mechanism, nested patterns create no overhead during the execution of applications.

Patterns with Runtime Behaviour The combination strategies presented up to here are all static combination strategies, i.e., extend or alter the structure of the generated pattern. Besides this type of combination strategy, the ϵSOA platform

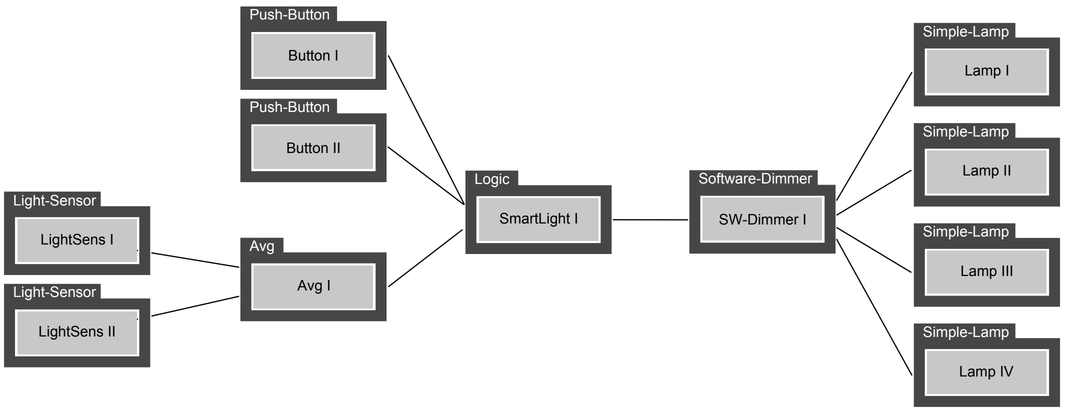
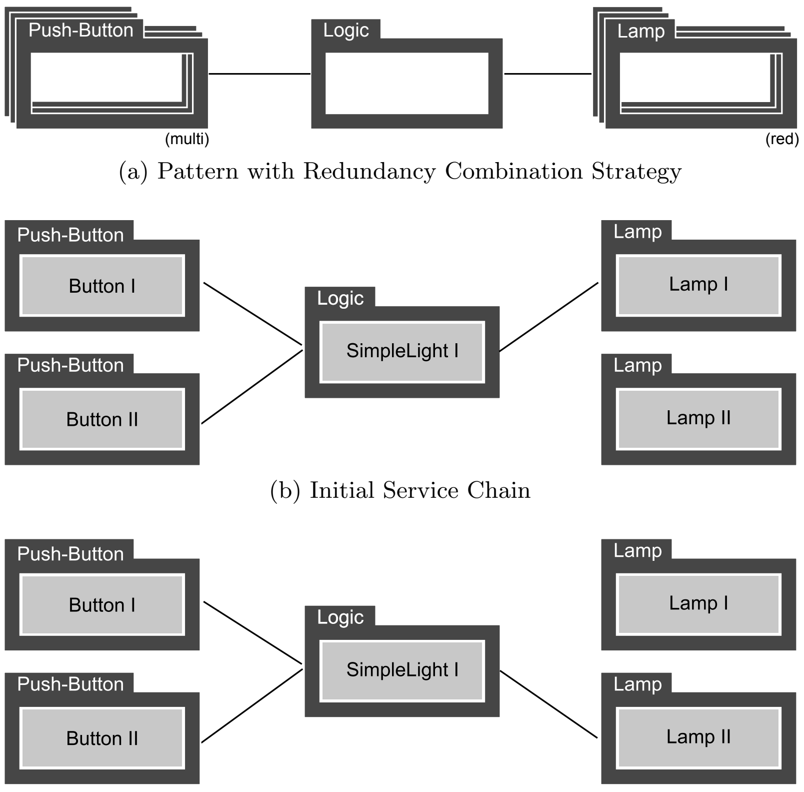


Figure 2.19: Service Chain using Combination Strategies and Nested Patterns



(a) Pattern with Redundancy Combination Strategy

(b) Initial Service Chain

(c) Service Chain after Failure

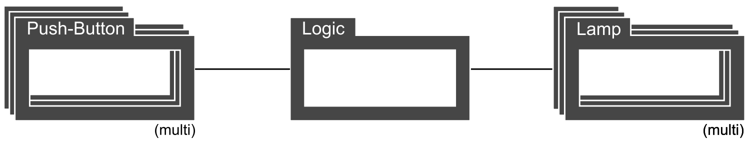
Figure 2.20: Redundancy Combination Strategy

also supports dynamic combination strategies that influence service compositions at runtime. One example for such a dynamic combination strategy is the “redundancy” combination strategy (red). Initially, the first service instance assigned to a slot with a redundancy combination strategy will be used. If this service fails at runtime, e.g., due to a node failure or energy depletion, it is replaced by one of the other instances assigned to the slot. This mechanism is especially useful if it is applied to logic services, because these can be installed at multiple nodes to provide the required redundancy. If multiple instances of a sensor and actuator device are available, the redundancy strategy can be applied to the corresponding hardware services, too. A possible pattern that uses redundancy is shown in Figure 2.20. Based on the services from the running example, the user can build the application shown in Figure 2.20. The upper service composition is active at rutime. If “Lamp I” fails, it is replaced by “Lamp II”. The resulting service composition is shown in the lower part of Figure 2.20. If the user repairs the “Lamp I”, one of two configurable actions is taken. If the user chose to use “Lamp I” as primary device, the system will reactivate the upper chain and use the replaced device. If the user selected no primary device, no changes are conducted and the replaced device will work as redundant backup for the lower device[11]. This “switching” between redundant service instances can be implemented efficiently by a simple controller running directly on the nodes in the embedded network. It can be performed fully automatically without user interaction and in a very timely manner. The failure compensation mechanism and some extensions are explained in more detail in Section 4.1.

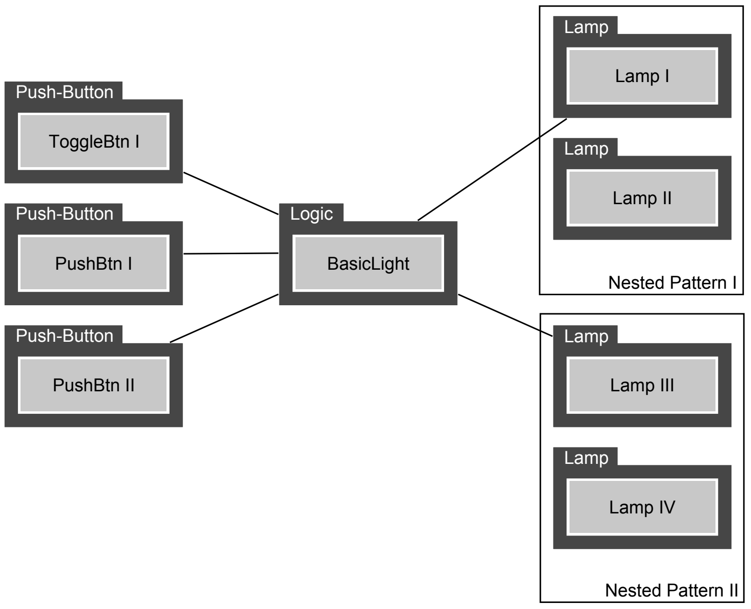
By using nested patterns, the redundancy combination strategy can also be combined with the “multi” strategy or one of the aggregation strategies mentioned in the previous examples. Assume that the lamps from the running example are very bright and two lamps are sufficient to light up the whole room. The user decides to build two groups of lamps. Each group contains two lamps which should be used redundantly, “Lamp I” and “Lamp II” in group one, and “Lamp III” and “Lamp IV” in group two. The user instantiates two nested patterns with a redundancy combination strategy and assigns the lamps to the corresponding slots. These nested patterns can be added to the pattern shown in Figure 2.21(a), which we already used in one of the previous examples. This pattern uses a “multi” combination strategy. The subpatterns containing the redundant lamp services can be assigned to the “Lamp” slot in the pattern, resulting in the service composition shown in Figure 2.21(b). The three buttons can be used to turn on or off “Lamp I” and “Lamp III” simultaneously. If one of these lamps fails, it is replaced by a redundantly available instance, “Lamp II” or “Lamp IV” respectively.

It is also possible to combine dynamic and static combination strategies the other way round, i.e., to define a static combination strategy as nested pattern, which is then added to a slot with a redundancy strategy. Nested patterns may also contain nested patterns again. As a consequence, all combinations of dynamic and static

[11]If multiple replacements are available for a redundant slot, candidates are selected in the order they were assigned to the slot or by user assigned priorities.



(a) Pattern



(b) Service Chain

Figure 2.21: Dynamic Combination Strategies and Nested Patterns

combination strategies can be created.

Dynamic combination strategies can access all monitoring information gathered on the nodes in the embedded network, e.g., remaining energy resources, utilization information, etc. Based on this information, further dynamic combination strategies can be developed. One example is to homogenize the energy utilization on battery powered nodes by switching between instances from different nodes based on the reported energy resources.

2.5.3 Summary

The ϵSOA platform offers pattern based service composition as alternative to a manual composition of services. Service patterns define an abstract service composition, which can be instantiated by assigning instances from a given installation to the placeholders defined in the pattern. The service placeholders used in the pattern define requirements that have to be fulfilled by services that should be assigned to the placeholder. The required functionality can be provided by different services in different systems. Patterns can be used to design reusable applications that can use whatever sensor/actuator device is present in a given installation - as long as this device provides the required functionality. At the same time, the application development with patterns is simple enough to be performed even by untrained users. This is achieved through a separation of the service/application developer, which creates the pattern and the used logic services, from the end-user, which only has to fill the pattern with services from a given installation. The assignment of services to placeholders in the pattern can be done based on domain knowledge and requires no further programming skills. The service descriptions and the requirements specified in the placeholders can thereby be used to esure that only meaningful service compositions are created, i.e., no incompatible services are linked together. Through the use of combination strategies, the flexibility and functionality of patterns can be improved. Combination strategies allow the definition of patterns that can work with arbitrary numbers of sensors and actuators and allow the automated injection of aggregation services into the resulting service compositions. Furthermore, combination strategies can be used to create applications that dynamically adapt to changes in the embedded networks, e.g., to compensate node failures through redundant devices or to adapt the execution of applications based on resource utilizations.

In some cases, service patterns can be used to automate the service composition process. Whenever there is only a single suitable candidate for a given placeholder in a pattern, the corresponding service instance can be assigned automatically. If multiple candidates qualify, the user has to perform the selection of the most suitable one. With a corresponding tool support, this mechanism allows a rapid installation and configuration of applications in embedded networks. An interesting functionality that can be provided by patterns is the simultaneous management of multiple pattern instances in large scale embedded networks. Consider a large office building. Many offices will be running applications that are based on the same pattern. A modification of one of these installations could be “copied” to similar installations in other rooms. We are currently investigating this and related reconfiguration mechanisms.

Another direction for future work are automated learning mechanisms for application patterns. The idea is to analyze the commonalities between the applications running in a given installation and to automatically derive general applications patterns. This could be used to support very dynamic development workflows. A user could create an application by composing services manually. Based on this composition, a general pattern could be derived (probably by comparing the given service composition with a repository of other compositions), which can then be used to “copy” the application to other rooms in a building.

2.6 Optimization Techniques

A benefit of the model driven development approach used in the ϵSOA platform is the explicit specification of application requirements in the application model. Based on this information, the configuration of the embedded network can be tuned to achieve an as efficient execution of applications as possible. In this work, we focus on two optimization fields: the optimization of measurement data rates and the optimization of the service placement, i.e., the location where services are executed. Besides these two fields, there are other optimization possibilities such as the re-use of datastreams in multiple applications and the optimization/configuration of communication links with TDMA schemes, which we will outline shortly at the end of this section.

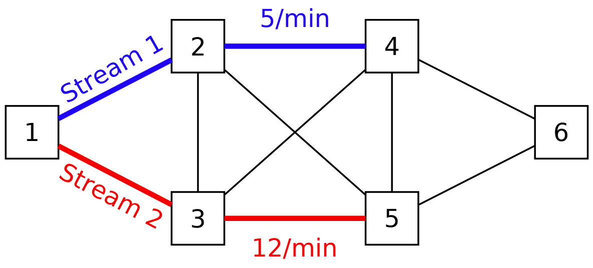
2.6.1 Data Rate Optimization

Sensor devices can be used by more than one application simultaneosuly. Each application has its own requirements concerning the data rate at which measurements should be delivered. The challenge is to coordinate the access to the sensor device in a way that ensures these requirements are met. Additionally, it is desirable to share measurements between applications. Depending on the sensor type, the activation of the sensing hardware can be an energy consuming operation. In addition to this, each measurement also incurs some processing overhead for analyzing the measured data. If the outgoing data streams of a sensor service are coordinated in a way that allows to use one measurement in multiple streams, a considerable amount of measurement operations can be saved. This leads to a reduction of the overall power consumption and increases the lifetime of battery powered nodes. The ϵSOA platform provides two means for achieving this goal: data rate ranges and a stream dispatcher module. Both concepts are explained in detail in the following sections.

2.6.1.1 Data Rate Ranges

The desired data for a data stream can be specified in the application pattern or added at installation time by the manager. In many cases an application has not one single acceptable data rate but a broader range of data rates that are acceptable. Consider for example a heating application. The control logic requires periodic temperature readings. These readings have to be frequent enough to react to short term temperature variations, e.g., an opened window. This is the lower bound for the data rate of temperature readings. There is also an upper bound, because a heating system is not very dynamic and increasing the measurement frequency beyond a certain point will not increase the quality of the resulting control operations.

In the ϵSOA platform data rates are specified by a triple comprising the minimum data rate, the maximum data rate and an optimal data rate. The former two values specify the range of acceptable values. The ϵSOA platform guarantees that the resulting data rate will lie in these bounds - as long as the sensor hardware supports



(a) Two Streams Created by One Sensor

Stream	Min.	Max.	Opt.
1 (blue)	3	7	5
2 (red)	10	14	12

(b) Data Rate Specifications (measurements per minute)

Figure 2.22: Header Generation for a Data Stream Transmission

the data rates of course. The optimal data rate can be used to indicate the sweet spot in this range. The ϵSOA platform will try to deliver data rates that are close to this value. However, deviations may occur if multiple data streams access the same sensor device. Note that the specification of the optimal data rate is optional. If a service performs equally well with any data rate laying in its acceptable range, the optimal data rate may be omitted.

The data ranges already provide a basic way of supporting multiple outgoing data streams with different data rates. If the intersection of these ranges is not empty, a data rate can be chosen that is acceptable for all applications. Of course this is not possible in general, because often data ranges will not overlap. In these cases it is still possible to achieve a sharing of measurements by using subsampling. Subsampling and the ϵSOA Stream Dispatcher, which performs this task, are introduced in the following section.

2.6.1.2 The ϵSOA Stream Dispatcher

Assume we have the situation depicted in Figure 2.22(a). A sensor on node 1 is used to create two data streams, Stream 1 (blue color in the figure) directed at node 4 and Stream 2 (red color) directed at node 5. Table 2.22(b) shows the corresponding data rate specifications. Stream 1 has an acceptable range of data rates between ${}^{3}\!/_{min}$ and ${}^{7}\!/_{min}$ measurements per minute with an optimum of ${}^{5}\!/_{min}$. Stream 2 a range of ${}^{10}\!/_{min}$ to ${}^{14}\!/_{min}$ with an optimum of ${}^{12}\!/_{min}$.

In this case, the data ranges of the individual streams do not overlap. Assume we measure data at a rate of ${}^{12}\!/_{min}$. This is the optimal data rate for Stream 2. If we take every second of these measurements and create a new data stream out of these values, we get a data stream with a data rate of ${}^{6}\!/_{min}$. This data rate is not optimal for Stream 1, but still in the acceptable range. This process is called subsampling. We created a new data stream by using a part (every second element) of the original data stream. Note that we are interested in creating subsamples that possess a fixed data rate. The subsample is always created by taking every i'th item of the stream,

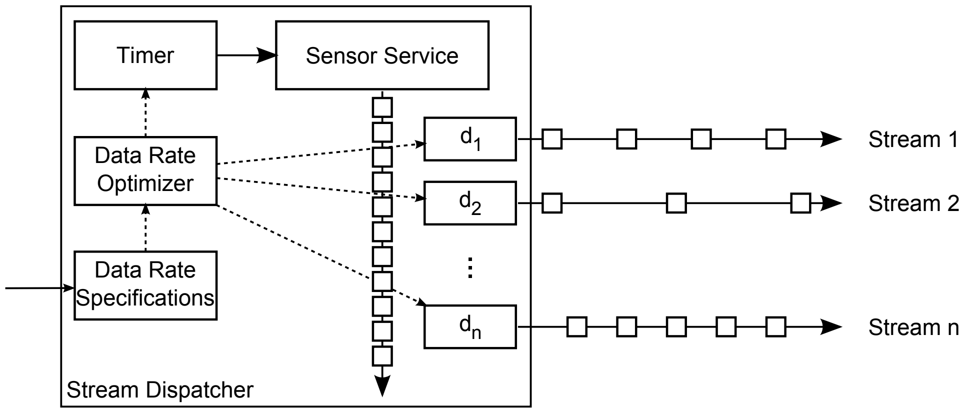


Figure 2.23: The ϵSOA Stream Dispatcher

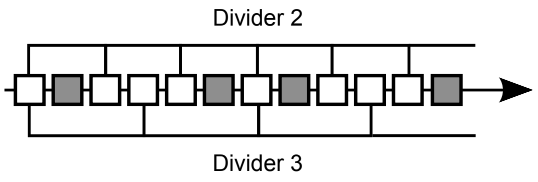


Figure 2.24: Skipping of Measurements

where $i \in \mathbb{N}, i > 1$. We will use the term divider $d = 1/i$ in the remaining part of the work[12].

In the ϵSOA platform, the subsampling is performed by the Stream Dispatcher. The architecture of the Stream Dispatcher is shown in Figure 2.23. The Stream Dispatcher controls the creation of data streams for a specific sensor, which is represented by a corresponding sensor service. The Stream Dispatcher has an internal timer that is used to periodically trigger measurements. This internal data stream is handed over to a series of splitting modules. Each splitting module creates an outgoing data stream i based on a supplied divider d_i. The splitting modules can be implemented very efficiently using a single counter that counts the number of dropped measurements. Whenever the counter reaches d_i, a measurement is added to the outgoing data stream and the counter is reset. The last two components of the Stream Dispatcher are a table holding data rate specifications for all outgoing streams, and an Optimizer that configures the internal timer and the splitting modules based on the given specifications.

To optimize the energy consumption, the internal Timer does not trigger the Sensor Service immediately. Instead it first checks whether any of the splitting

[12]Note that the term subsampling is also used in statistics whenever a smaller sample is taken from an existing sample. In this case, not necessarily every i'th element will be selected.

modules will create an outgoing data packet or not. If that is not the case, the sensing hardware is not activated. Instead a dummy measurement is added to the internal data stream. The dummy measurement will cause the splitting modules to correctly increment their counters. This optimization saves unnecessary measurements that can occur if not all elements of the internal stream are forwarded to an outgoing stream. A simple example scenario are two outgoing data streams with dividers 2 and 3, as seen in Figure 2.24. In this case, out of six elements of the internal stream, the first element is used in both streams, the third and fifth only in the stream with divider 2 and the forth only in the stream with divider 3. The second and the sixth element are not used in either stream (indicated by the gray boxes) and can be replaced by dummy measurements.

Subsampling has two major benefits. First measurements can be triggered with a single periodic timer. This is beneficial compared to a solution using multiple timers which have to be coordinated to avoid conflicting accesses to the sensing hardware. Second subsampling can reduce the number of measurements taken by sharing measurements between multiple streams. In the example from Figure 2.22(a), we needed only a measurement rate of $^{12}/_{min}$ to create both data streams, because Stream 1 re-used every second measurement from stream 2. If both streams had been created independently, a total of $^{12}/_{min} + ^{5}/_{min} = ^{17}/_{min}$ measurements per minute would have been required. In this case, the timers for stream 1 and 2 would collide every 60 seconds. Even if we could exploit this fact and share this measurement between both streams, we would still require a total of $^{16}/_{min}$ measurements per minute. Compared to the $^{12}/_{min}$ measurements per minute needed with subsampling, this is still a considerable increase. On the other hand, the optimal data rate of stream 1 is not met perfectly when using subsampling. A trade-off has to be found that allows reducing the number of measurements taken, but still provides data rates that are close to the desired optima. This trade-off is performed by the Data Rate Optimizer contained in the Stream Dispatcher.

2.6.1.3 Optimization of Data Rates

We already introduced the conflicting goals encountered by the Stream Dispatcher with an example in the previous section. We will reformulate the problem more mathematically.

Optimization Problem The Stream Dispatcher has to solve the following optimization problem: given a set of n data streams with data rate optima $o_1, o_2, \ldots, o_n$, find an internal measurement data rate r and a set of dividers $d_1, d_2, \ldots, d_n$ that minimzes the deviations from the data rate optima and minimizes the number of measurements taken per minute.

Because we supplied two optimization criteria, the optimization problem has multiple solutions. It is always possible to find a measurement rate and a set of dividers that perfectly suites all data rate optima. But this solution will have a very high number of measurements, because only few measurements are shared. The other extreme is to use a very low measurement rate and accept high deviations from the optima. To combine both optimization criteria, we use a weighting function w, which is defined as:

$$w = c \cdot dist + (1 - c) \cdot u$$

$dist$ quantifies the deviation from the optima and u the number of required measurements respectively. This weighting function allows finding a good trade-off between the deviation from the optima and the required number of measurements. We used a value of $c = 0.5$ for our experiments. An increase in the deviation from the optima of $x\%$ is therefore tolerable, if it results in a decrease of the number of required measurements of at least $x\%$.

Distance Metric The resulting data rate for each stream is the d_i'th part of the measurement rate r. $dist$ quantifies the deviation from the optima as a percentage value. It is defined as:

$$dist = \sum_{i=1}^{n} \left| \frac{r}{d_i} - o_i \right| \cdot \frac{1}{o_i}$$

where o_i is the optimal data rate for stream i.

Number of Measurements The calculation of u is more complicated. A solution without any sharing would require a total of $\sum_{i=1}^{n} o_i$ measurements. We are only interested in solutions that require less measurements than this base case. We therefore define u as the savings achieved compared to this base case:

$$u = \frac{m}{\sum_{i=1}^{n} o_i}$$

where m is the number of measurements taken per minute.

In general m is not equal to the data rate r. As exemplified in the previous section, some readings can be omitted because they will be ignored by all splitting modules. As a consequence m is typically smaller than r. Only if a divider of 1 is used, m is equal to r.

Let A_i be the set of measurements acquired in one minute that are used for creating output stream i. The number of required measurements per minute m is then:

$$m = \left| \bigcup_{i=1}^{n} A_i \right|$$

The cardinality of A_i is easy to determine. It is the d_i'th part of the internal measurement rate r, thus

$$A_i = \frac{r}{d_i}$$

The sets A_i are not disjunct. In order to determine the cardinality of the union of these sets, the inclusion-exclusion principle can be used. The inclusion-exclusion principle is used in combinatorial mathematics to determine the cardinality of a

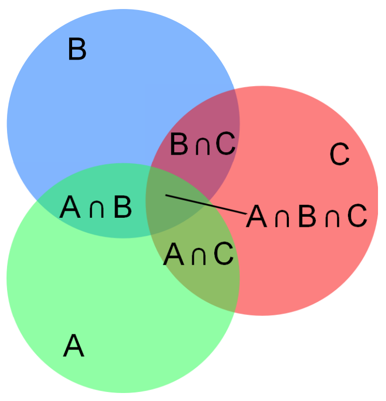


Figure 2.25: Principle of Inclusion-Exclusion

union of finite non-disjunct sets. The basic idea is to calculate the cardinality of a union by first using a over-generous inclusion, which is followed by an exclusion to compensate the error. Figure 2.25 shows an example comprising three sets A, B and C. Following the inclusion-exclusion principle, the cardinality of $A \cup B \cup C$ is calculated by starting with the over-generous inclusion $|A| + |B| + |C|$. This inclusion was over-generous, because we counted the elements in the intersection multiple times. We correct this error by substracting the elements contained in the intersections, which are $|A \cap B|, |A \cap C|, |B \cap C|$. This was an over-generous inclusion again, because we included the elements contained in $A \cap B \cap C$ multiple times. This error is compensated by adding another exclusion, and so on. For the special case of three sets used in the example, the resulting cardinality is

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$$

For the general case, the cardinality is

$$\begin{aligned} m = \left|\bigcup_{i=1}^{n} A_i\right| = & \sum_{i=1}^{n} |A_i| - \\ & \sum_{i,j\,:\,1 \le i < j \le n} |A_i \cap A_j| \qquad + \\ & \sum_{i,j,k\,:\,1 \le i < j < k \le n} |A_i \cap A_j \cap A_k| - \ \cdots \ + \\ & (-1)^{n-1} \, |A_1 \cap \cdots \cap A_n| \end{aligned}$$

In order to calculate m, we first have to determine the cardinality of the various possible intersections. Given these values, we can determine the value of m using the inclusion-exclusion principle. The measurements that lay in the intersection

Stream	Min.	Max.	Opt.
1	40	60	50
2	15	25	20
3	12	24	18

(a) Example Scenario

d_1	1	1	1
d_2	3	2	2
d_3	3	3	2
r	50	50	40
$dist$	0.241	0.324	0.311
u	0.568	0.568	0.455
w	0.404	0.446	0.383

(b) Data Rate Specifications (measurements per minute)

Figure 2.26: Example Calculation

of two streams are the measurements that are (re-)used by both these streams. Assume the measurements in the internal stream are numbered in increasing order. A measurement is used in both streams, whenever the measurement's number is a multiple of the dividers of both streams. The rate at which these collisions occur is therefore the least commom multiple (LCM) of the dividers.

The number of required measurements per minute is then:

$$\begin{aligned} m = \left|\bigcup_{i=1}^{n} A_i\right| = \; & r \cdot \left(\sum_{i=1}^{n} \frac{1}{d_i} - \right. \\ & \sum_{i,j\,:\,1 \le i < j \le n} \frac{1}{LCM(d_i, d_j)} \quad + \\ & \sum_{i,j,k\,:\,1 \le i < j < k \le n} \frac{1}{LCM(d_i, d_j, d_k)} - \cdots + \\ & \left. (-1)^{n-1} \frac{1}{LCM(d_i, \ldots, d_n)} \right) \end{aligned}$$

Based on this result for m, the number of required measurements u can be calculated using the formula above.

Algorithm The algorithm for solving the optimization problem iteratively tries different sets of dividers. For each set, it calculates the optimal measurement rate r and the values for $dist$ and u. The algorithm stores the best solution w.r.t the weighting function w.

Assume we have three streams with the data rate optima specified in Table 2.26(a). Table 2.26(b) shows the calculation results for some combinations of dividers. We will go through the calculation of the first set of dividers from Table 2.26(b) step by step.

The first step is the calculation of r. It is performed based on the distance metric. The goal is to find a value for r that minimizes *dist*, i.e., that minimizes:

$$dist = \sum_{i=1}^{n} \left| \frac{r}{d_i} - o_i \right| \cdot \frac{1}{o_i}$$

The absolute values in the formula above can be eliminated by using case differentiations. The sign of the term inside the absolute value flips at $\frac{r}{d_i} - o_i = 0$, what is equal to $r = o_i \cdot d_i$. We can now distinguish $i + 1$ different ranges for r. The first range are data rates below 50. The second range are data rates between 50 and 54, the third range between 54 and 60. Finally the fourth and last range are data rates greater than 60. For each of these ranges we calculate the data rate r that minimizes *dist*. This can be done easily, because the absolute values can be removed. We select the data rate r that minimzes *dist*.

For the example, this is a data rate of 50, which results in a distance of 0.241. The resulting data rate for stream 1 is 50, which fits perfectly. Stream 2 has a data rate of $^{50}/_{3} \approx 17$ which shows a comparably large deviation from the optimum of 20. Stream 3 has the same data rate, which is fairly close to the optimal value of 18. Summing up, we have a deviation *dist* of approximately 24% from the optimal data rates.

Based on r and the dividers, we can calculate u. In the example, u is 0.568. We therefore only need approximately 57% of the measurements compared to the simple solution using multiple timers. These saving are achieved, because all the measurements used in streams 2 and 3 are shared with stream 1. The resulting value of w is 0.404.

As we can see from the table, this value is quite good but not optimal. A better solution can be found using the dividers $2, 5, 6$ and a data rate of 100. In this case, the deviation from the optima is much smaller. Stream 1 is supplied with a data rate of $^{100}/_{2} = 50$, which fits perfectly. The same holds for Stream 2 with a data rate of $^{100}/_{5} = 20$. Stream 3 has a data rate of $^{100}/_{6} \approx 16.7$, which is fairly close to the optimal value of 18. Given these values, *dist* is 0.074. The resulting value for u is 0.682, what is slightly worse than the value for the dividers $1, 3, 3$. The savings gained in *dist* overweigh the increase in u, the solution using the dividers $2, 5, 6$ is therefore superior. It is even the best solution that can be found.

Search Space The number of reasonable divider combinations is limited. The optimization algorithm iteratively selects a fixed divider d_l for the largest data rate optimum, e.g., the value of 1 for Stream 1 in the example above. In order to minimze the deviation from the optima, the ratio between the other dividers and 1 should resemble the ratio between the corresponding data rate optima. In the example the ratio between the optima of Stream 2 and Stream 1 is $^{50}/_{20} = 2.5$. The ratio between the dividers should be close to this value, leaving possible divider values of 2 and 3 for Stream 2. This results in a total of 2^{i-1} combinations for each value of d_l.

In a practical setting, the search space for d_l is restricted by u. Assume we use a divider value of $d_l = 4$. Because d_l is used for the stream with the highest data rate, all other dividers have to be equal or greater than 4, too (the ratio between the largest optimum and the other optima is always greater or equal to one). High divider values result in a low number of shared measurements, because the LCM grows very fast. In a practical setting, divider values d_l lower or equal to 4 are typically sufficient. The resulting number of possible combinations can be tested even on severely constrained microcontrollers.

2.6.2 Service Placement

Services that do not depend on the presence of a specific hardware (e.g. a sensor or actuator device) can be executed on any node in the embedded network, assuming the node posseses the memory and processing resources required by the service. This flexibility is a central building block for the design of scalable and robust distributed control applications. It allows remedying overload situations by moving services from the overloaded node to other nodes in the network, and allows adapting the execution of applications based on the energy resources of the underlying nodes in order to homogenize the battery utilization and move services to neighboring nodes if an energy depletion is forseeable. Furthermore, the scalability of embedded networks can be increased. If the summed processing power in an embedded network is not sufficient, a new node can be added and some of the already installed services can be moved to this node.

There are boundary conditions that have to be taken into account during these optimizations. A service can only be moved between nodes if the target node possesses the required hardware devices (important for sensor/actuator devices) and enough resources to execute the services. These constraints can be modeled in the system model in the ϵSOA platform. The optimization algorithm presented in this section can derive an optimized service placement, i.e., an optimized assignment of services to nodes, based on such constraints, the specified application requirements and the network infrastructure. The new placement can be viewed and modified by the user using the tools presented in Chapter 5 and finally deployed in the embedded network.

On the following pages, we will first present the service placement optimization problem in Section 2.6.2.1. After that, we will introduce a set of metrics in Section 2.6.2.2 that allow quantifying the quality of a specific placement. These metrics are used by optimization algorithms, which we will present in Section 2.6.2.4, to calculate an optimal service placement for a given set of applications and network characteristics. We performed a series of benchmarks to compare different optimization techniques and to evaluate different configuration parameters of the optimization algorithms. The network generator used in these tests is described in Section 2.6.2.3. The benchmark results are presented in Section 2.6.3. Finally, we will present an availability metric in Section 2.6.4, which can be used to place services in a way that maximizes the availability of the network in case of node failures.

2.6.2.1 The Service Placement Optimization Problem

The quality of a placement is defined by a set of metrics which are defined in the following section. The optimization problem is to determine a mapping of services to nodes that creates minimal costs w.r.t. these metrics and fulfils a set of boundary conditions. A placement is defined as a function $p(i) = n, i \in I, n \in N$ that maps every instance i out of the set of all instances I to a node n out of the set of all nodes N.

Resource Constraints The first boundary condition for every placement are the resource constraints on each node. Every service requires a specific amount of resources $R = \{r_1, \ldots, r_n\}$ (we currently use CPU, flash memory and RAM) on a node. The sum of the demands on a node may not exceed the capacity of this node. Each service instance i specifies a resource demand vector $d_i = (d_{r_1}, \ldots, d_{r_n})$ which quantifies the resource demands for every resources $r_i \in R$. Each node n specifies a resource capacity vector $c_n = (c_{r_1}, \ldots, c_{r_n})$ for every resource $r_i \in R$. Every placement has to fulfill the following boundary condition:

$$\forall_{n \in N} \forall_{r_i \in R} \left(\sum_{i \in I, p(i)=n} d_i(r_i) \right) < c_{r_i}$$

Assignment Restrictions Another boundary condition are assignment restrictions. Some service instances can only be installed at nodes that possess some specific hardware, e.g., a specific sensor device. Another source of restrictions are administrative restrictions, e.g., some service may only be installed at nodes in a specific location etc. These restrictions are notated for a service instance i as a set of suitable nodes $s_i \subset N$ where N is the set of all nodes. Note that assignment restrictions can be used to create non-movable service instances, i.e., instances which may not be relocated to another node, by specifying only a single target node in the list of suitable nodes.

Nodes With Limited Reprogramming Capabilities Depending on the hardware and the used operating system, nodes may be reprogrammable at runtime or not. If a node is reprogrammable, it is capable of executing any service instance as long as it has enough spare resources and possesses the hardware devices required by this service instance. If the node is not reprogrammalbe, e.g., because the operating system does not support the dynamic loading of code, it can only execute the services that were initially installed on this node. These restrictions are modeled by specifying a list of possible services that may be executed on a node.

2.6.2.2 Metrics

A prerequisite for the calculation of a service placement are metrics that allow to quantify the quality of a placement and allow comparing different placements. The available metrics depend on the information available in the system model. For the calculation of some of the metrics mentioned in this section, information from the routing layer is required to determine the routes used in the physical network for the transmission of data streams (the streams only specify the start and end point of the transmissions, not the hops in between). This information can be either supplied by an algorithm that models the behaviour of the routing protocol and calculates the shortest path between nodes in the physical network, or can be queried from the network protocol (using a suitable cross layer communication mechanism, like the one described in Section 3.8).

At the current stage, we do not support timing constraints during the calculation of metrics. For some metrics, such as the CPU utilization, it is not only important how large the demand of a service for this resource is, but also *when* it is requested. If the underlying resources can only be used exclusively, simultaneous demands will result in delays and increase the time needed to execute a service. We are currently investigating how the execution model on the nodes and the timing requirements of the services can be incorporated to improve the calculation of the metrics described in the following paragraphs.

Many of the metrics described below require information from the system model regarding the capabilities of the available hardware and the requirements of the applications. In many cases, this information will be available immediately because it is contained in the hardware specifications or given by the application developer. If this is not the case, a lot of information can also be collected at runtime by observing the service execution on the nodes. In this scenario the system will be launched with a placement based on a very simple metric, e.g., the hop count, and can be optimized when additional information is available through monitoring.

Currently we have implemented 6 metrics. Just like new properties in the system model, new metrics can be added easily to the system to allow a customization for specific application fields. For the utilization metrics, we will describe how the utilization coefficient for every node is calculated. These coefficients are combined to receive the overall utilization based on the maximum, mean or a specific percentile of the coefficients.

Hop Count A simple metric that is always available is the hop-count. For the calculation of this metric, the number of hops involved for the transmission of all data streams flowing through the system is summed up. This very simple metric works fairly well for the optimization of the network utilization if the data streams used by the applications have similar data volumes.

Data Volume If information about the expected volume of data-streams is available, the hop-count metric can be refined to calculate the data volume metric. This metric is based on the summed data volume transmitted over all links, i.e., the data rate of each stream multiplied with the number of hops needed for routing the stream. If an application comprises services that produce low data volume streams out of high data volume streams, e.g., a control service like the one presented in the air condition example that requires periodic measurements but only rarely issues commands to an actor service, this metric will ensure that the data consuming service is placed as close to the data producing services as possible (preferably on the same node). This metric closely resembles the heuristics used in systems like TinyDB, which "push" services as close to the stream sources as possible.

Link Utilization The data volume metric can be further extended to calculate the overall link utilization metric, if additional information about the bandwidth of the links is available. For each link the summed data rates of all streams flowing through a link is divided by the link's bandwidth. If this coefficient is greater than one, the link is marked as overloaded[13].

Network Utilization In many cases, logical links to different nodes are using the same physical communication medium, e.g., a wireless link or an ethernet link which is connected to a switch. To avoid overload situations under these circumstances, an additional utilization metric, the network utilization is calculated for each physical communication medium available at each node. This is done by aggregating the data volumes for all logical links using the same physical medium, e.g., all ZigBee links.

Memory Utilization The memory utilization metric can be calculated if information about the memory demand of services is available. In many cases this information can be determined by inspecting the service code. If this is not the case, the user has to specify the corresponding value manually in the system model or it has to be determined at runtime by observing the memory usage of the running service.

CPU Utilization The CPU utilization is calculated based on CPU cycles. The calculation of this metric requires information about the CPU capacity of each node, and an estimation of the required CPU cycles for the execution of each service. To estimate/measure the number of required CPU cycles, emulators such as (Power) TOSSIM[91, 137], MSPSim[36] or similar tools for other operating systems and hardware platforms can be used. A counter for the number of CPU cycles used in each service invocation (which relates to a method invocation performed by a dispatcher when a new message is received or a periodic timer is triggered) can be easily implemented in these tools or is often already available because it is required to calculate the CPU power consumption. The emulation run can be performed by

[13]Placements containing overloaded resources are not immediately discarded because the user can opt to install the applications based on these placements anyway. This can be a reasonable decision if all placements result in overload situations and the middleware possesses features to compensate link congestions at runtime, e.g., by dynamically reducing the data acquisition rates at the sensor devices.

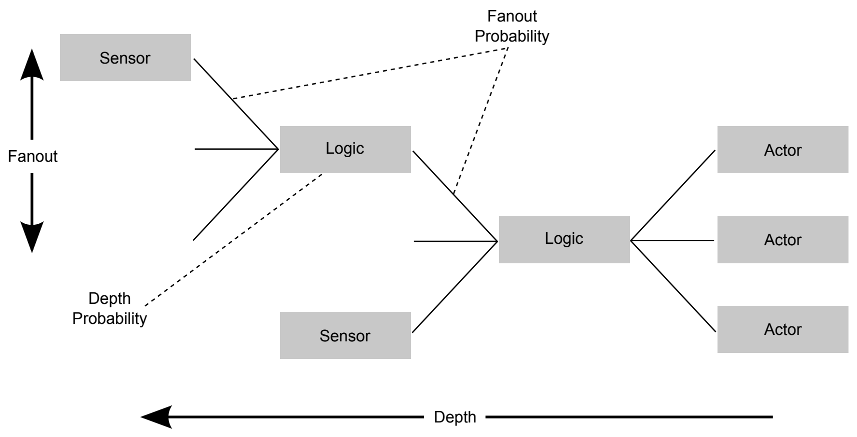


Figure 2.27: Basic Structure of Generated Applications

the service developer prior to publishing the service in a service repository and the results can be annotated in the service description[14].

Combined Metrics The individual metrics mentioned above can be combined with a linear weighting function. The weight for each metric can be specified by the user. This mechanism can also be used to create a list of alternative solutions. In this case, the optimization algorithm is invoked multiple times with different weights prioritizing different metrics. The user can browse this list of alternatives using the development tools presented in Chapter 5 and select the most suitable one. In ongoing work, we are studying how the generation of alternative solutions can be refined to create a set of pareto optimal solutions. A possible approach is outlined in the description of ongoing work at the end of this section.

2.6.2.3 Evaluation Scenario

In order to get a large test set of different embedded networks for the testing of the optimization algorithms we designed a system generator, which creates network and application models based on a set of parameters.

Network Structure The created networks are based on a grid based structure. Each node in the grid has a communication link to each of its four adjacent nodes with a configurable probability (if the generated network is non-connected, it is discarded and a new network is generated). The generator can also be used to create link characteristics (e.g. bandwidth limitations) in a user specified range.

Application Model The generator can create a configurable number of applications with different structure and complexity. Applications are created following the basic

[14]Such an emulator run is often performed anyway to do unit testing on an emulated device.

Scenario Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Number of Nodes	3x3	5x5	7x7	9x9	3x3	5x5	7x7	9x9	3x3	5x5	7x7	9x9	3x3	5x5	7x7	9x9	3x3	5x5	7x7	9x9
Number of Apps	2	2	2	2	4	4	4	4	2	2	2	2	4	4	4	4	6	6	6	6
Depth	3	3	3	3	3	3	3	3	4	4	4	4	4	4	4	4	4	4	4	4

Table 2.1: Evaluation Scenario for Service Placement Algorithms

structure shown in Figure 2.27. The generation starts with a logic service. To the output(s) of the logic service 0..*Fanout* actor services are attached, each with a probability of *FanoutProbability*. Analogously, 0..*Fanout* services are attached to the inputs of the logic service, again with a probability of *FanoutProbability*. These service can either be sensor services or logic services. The latter are created with a probability of *DepthProbability*. If a logic service is created, the creation recursively starts at this service, i.e., a set of service (either sensors of logic services) are attached to the input(s) of the logic service. The depth of the recursion, and therefore the length of the service chain, can be capped with the *depth* parameter. The sensor and actuator services are fixed at randomly chosen nodes in the network, the logic services can be placed freely.

Scenarios The following parameters were used for the evaluation: a maximum *Fanout* of 3, a *FanoutProbability* of 80%, a *DepthProbability* of 80% (i.e., 80% probability for the creation of a logic service). The remaining parameters were configured according to Table 2.1 to generate a series of scenarios with varying application complexity and number and different network sizes. This configuration resembles typical application sizes/structures we encountered during the development of our demonstrators.

2.6.2.4 Optimization Techniques

The task of the optimization techniques presented in this section is to determine an optimal placement, i.e., a placement with as little costs as possible, based on the metrics presented in the previous section and the generated system model containing information about the hardware characteristics and the application requirements. The optimization problem of distributing services to nodes can be easily mapped to the Bin Packing Problem: the task is to distribute n services with resource demands $d_1 \dots d_n$ to m nodes with resource capacities $c_1 \dots c_m$ in a way that avoids overload situations. The problem is therefore NP hard. For small networks (< 10 nodes) and a small number of services (< 10 services), a solution based on a simple enumeration of all possible combinations is possible. For larger problem instances, other solutions have to be applied.

We analyzed three well known optimization algorithms: Ant Colony Optimization, Simulated Annealing and a Genetic Algorithm, which we will present in the following sections. All algorithms are intended to be used on a central management node in the network that possesses global knowledge about the network topology, hardware characteristics and service requirements. This is typically the case for management nodes, which control the application execution in an embedded network or a subnet of a larger network. The algorithms aim at finding a global solution to the optimization problem, i.e., will move already installed services in the network if a new application should be installed and requires already occupied resources. These reorganizations come at a cost, because services have to be migrated between nodes and the corresponding applications will cease to work during the migration process. To provide a good trade-off between the migration costs and the long term savings of a new placement, the optimization algorithms can be used to create a list of placements containing different levels of reorganization. The user can then select an appropriate placement from this list. This is done by running the optimization algorithms multiple times with different restrictions for the placement of services, e.g., restricting all installed services to the node they are executed on will result in a scenario with no reorganization.

Ant Conoly Optimization Ant Colony Optimization (ACO)[20, 34] is inspired by the way ants explore food sources and optimize the track between the food source and the colony. (Initially) ants wander randomly. Whenever an ant finds food it leaves a pheromone trail on its way back to the colony. Other ants that cross such a trail will likely follow it and also leave pheronomes on their way back to the colony, therefore reinforcing the trail.

Simulated Annealing Simulated Annealing (SA)[14, 81] mimics the process of annealing used in metallurgy. The SA algorithm iteratively replaces the current solution with a “neighboring” solution, which is chosen based on the improvement (or the decrement) of the quality solution and a global parameter, the temperature. The temperature initially starts at a high value and decreases over time. When the temperature is high, worse solutions are accepted with a high probability (resulting in a comparatively random search in the optimization space at the beginning). Lower temperature values enforce a more “downhill” oriented behaviour, i.e., the algorithms will tend to accept only solutions that are actually better than the current one. The possibility to intermediately accept worse solutions and enforce a convergence only at low temperature values reduces the chance that SA gets stuck in a local optima, which is a typical problem for heuristics such as hill climbing (see e.g. Russel and Norvig[127]) or similar methods.

Genetic Algorithm A Genetic Algorithms (GA) is based on the idea of iteratively evolving a set of genomes (the population) in order to create better and better solutions to the optimization problem. Mapped to the service placement problem, we try to iteratively improve a set of service placements until we find an optimal solution to the optimization problem. Genetic algorithms use two basic concepts for evolving genomes: mutation and crossover. Mutation introduces some random changes to a genome, crossover creates a new genome based on a combination of characteristics from two parent genomes. As mutation function we use the same mechanism as in the neighbourhood function used in simulated annealing: we randomly move one service instance from one node to another. The implementation used for the tests below does not use crossing. The genetic algorithm iteratively creates a new generation of genomes (also called a new generation) out of the current one by mutating and crossing genomes. From the new generation, the s best genomes are selected and used as starting point for the next iteration. The parameter s is called the size of the gene pool.

Comparison of Optimization Techniques Ant Colony Optimization is not well suited for the optimization of the service placement problem. Many metrics used for the service placement optimization create a situation where multiple data streams compete for the use of a communication link (e.g. link utilization metric does). In other words, the decision to transmit a stream over a specific link (which is locally optimal for a specific application) can lead to a non-optimal global solution, because other applications are prohibited from using this link. In order to use ACO in such a situation, one would have to introduce an additional mechanism that modifes parts of the pheromone trails if a competition is detected. Such modifications have to be studied carefully to ensure they do not jeopardize the convergence of the algorithm towards the global optimum. We therefore chose to not use ACO.

We implemented both, Simulated Annealing and a Genetic Algorithm, and performed some initial tests to determine which optimization technique is better suited for solving the service placement optimization problem. These initial tests showed that - given the same CPU time - SA yields considerably better placements than our GA implementation. We therefore chose to analyse SA in more detail and to use an optimizer based on SA for the prototypical implementation of the ϵSOA platform. There is some potential to improve the genetic algorithm used for these initial tests. In ongoing work, we are investigating what performance gains are achievable and whether an improved implementation of the genetic algorithm can compete with the SA based optimizer (or even provide better results). We will present the possible improvements for the genetic algorithm and the results of the initial benchmark in

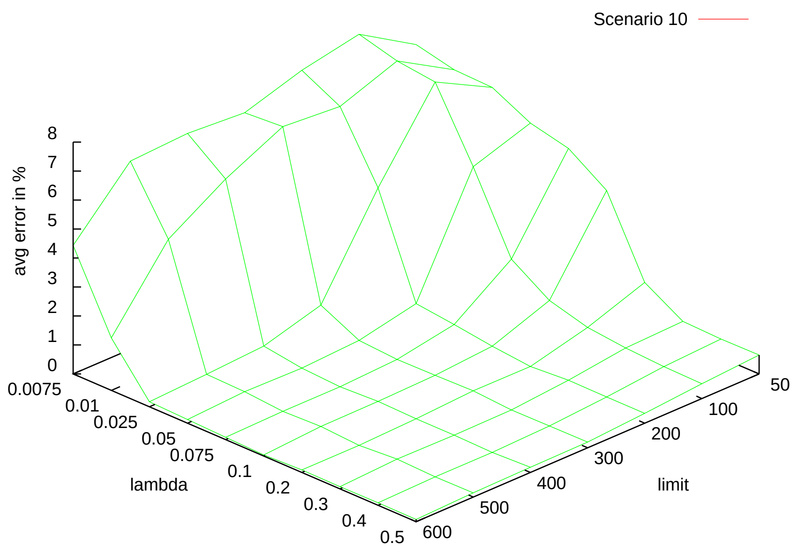


Figure 2.28: Lambda / Limit Test for Scenario 10

more detail at the end of this section. In the following section we will focus on the SA based implementation of the optimizer.

2.6.3 Service Placement Optimization with Simulated Annealing

Our implementation of Simulated Annealing is based on the algorithm from Russel and Norvig[127]. It uses an exponentially decaying temperature function $e^{-\lambda \cdot t}$, where t is the time counted as number of SA iterations performed so far. The algorithm is stopped after l iterations (also called "limit" in this work). The neighbor function used to generate new solutions moves a single service to another, randomly selected, node in the network.

2.6.3.1 Temperature Function Parameterization

We performed a series of tests to derive good values for the parameters used in the temperature functions, that is λ and l. For each scenario described in the previous section 50 different systems and a random starting placement were generated. Each of these systems was optimized using the SA algorithm with different settings for λ (ranging from 0.0075 to 0.5) and l (ranging from 50 to 600), resulting in a total of 35,000 test runs. The test runs were performed using the hop-count metric, as this metric is the fastest to compute[15]. Because the SA algorithm operates independently

[15]Which metric performs best for a given embedded network depends on the application scenario and the network characteristics (the utilization metrics for example are of little value if the

of the used metric, the results are applicable in settings with other metrics, too. Figure 2.28 shows for each parameter combination the average deviation from the "optimum" in percent for Scenario 10. The results for the other scenarios can be found in Appendix A.

The deviation is calculated as distance from the best solution found by the SA algorithm. Of course it is not guaranteed that this solution actually is the optimal one. Due to the high number of possible placement it is practically infeasible to calculate the optimum by enumerating all possible combinations for the more complex scenarios. We cross-checked the results of the SA algorithm with the results from a complete enumeration for the simple scenarios (3x3 grid) to check the correctness of the SA algorithm. In these cases, SA was always capable of finding the optimal solution. There is an indicator that this observation also holds for the larger scenarios (or that the found results are at least very close to the optimum): each scenario is analyzed by 70 runs of the SA algorithm (with the aforementioned different parameter settings). Due to the involved randomness, it is unlikely that the SA algorithm will repeatedly converge to the same local optimum for each of these runs. As a consequence, the probability is quite high that the best found solution actually is the optimal solution in the larger scenarios (or at least very close to the optimum).

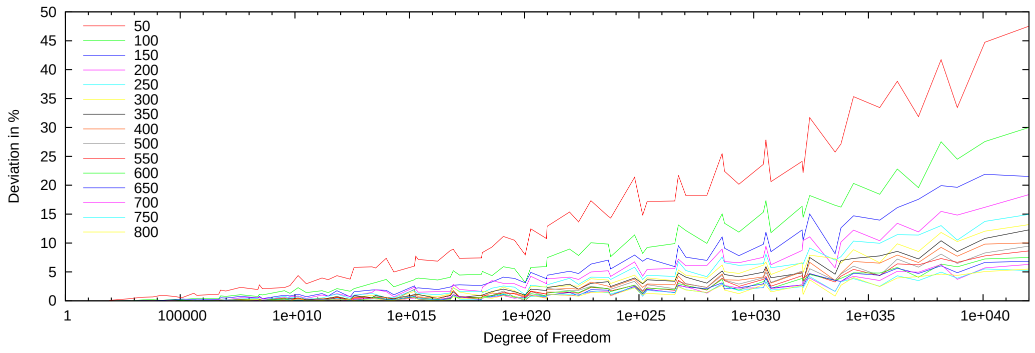
As Figure 2.28 shows, the SA algorithm is quite insensitive to the selection of λ. For very low values, and therefore high temperature values, the convergence towards the optimum is not strong enough. For very high values, the SA algorithms tends to converge to local optimas. This effect can be observed for the scenarios with a high number of applications (Scenarios 17-20)[16]. To achieve a good compromise between both situations we selected a value of $\lambda = 0.2$ which provides good overall performance. The benchmarks also show a clear dependency between the number of iterations, l, and the quality of the resulting placement.

2.6.3.2 Comparison of Neighborhood Functions

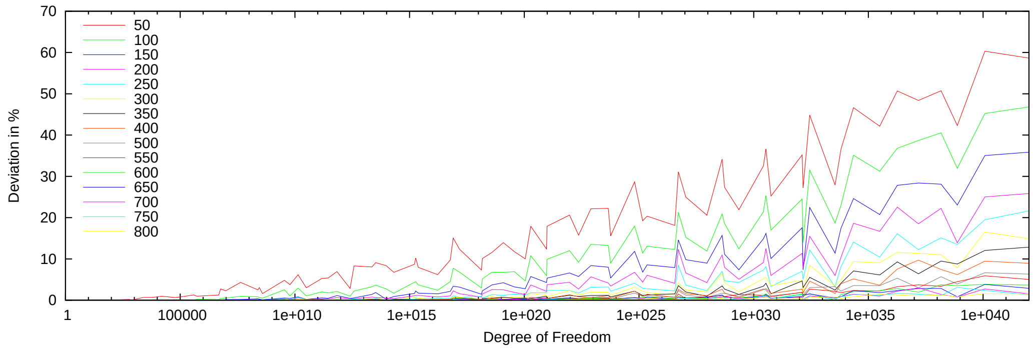
We ran another series of tests to further analyze the dependency between the number of iterations and the complexity of the scenario. These tests were also used to evaluate different neighbor functions: (1) the function used in the previous tests that moves a service to a random node in the network, (2) a function that randomly moves a service to adjacent nodes, and (2) a function that moves services to nodes that are two hops away in the network. The tests were run using the previously derived value of 0.2 for λ. For the tests, 50 systems were generated for each of the 20 scenarios and supplied to the SA algorithm using each of the above mentioned neighbour functions. An additional number of 50 systems was generated for Scenarios 18 and 19 and an additional number of 350 systems for Scenario 20 to get a reasonable number of repetitions for the larger systems. For each of these systems, the three

network bandwidth is high and the transmitted data volume is low). In this work, we therefore focus on the optimization algorithm itself. The comparison of the different metrics has to be performed depending on a concrete application scenario and is not in the focus of this work.

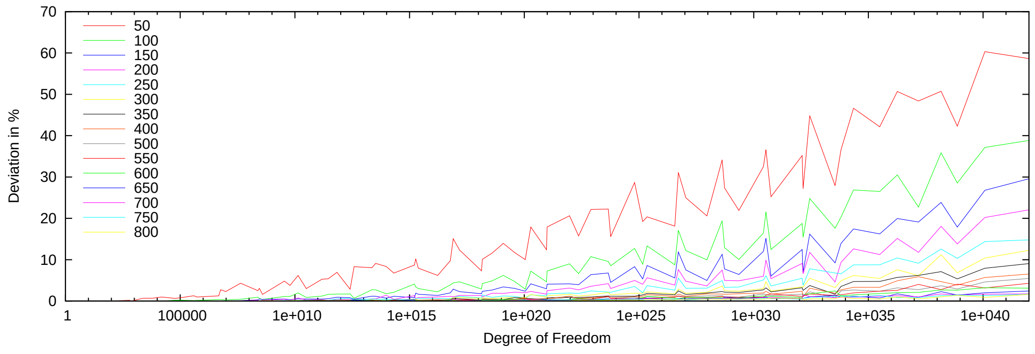
[16]The effect is hard to see in the graphics due to the scaling of the vertical axis.



(a) Neighbor Function: Random



(b) Neighbor Function: Adjacent Neighbor



(c) Neighbor Function: 2 Hops Distance

Figure 2.29: Deviation for Varying Degrees of Freedom

different neighborhood functions were tested with a varying number of iterations between 50 and 800, incremented in steps of 50. This results in a total of $3 \cdot 16 = 48$ SA runs for each system and a summed total of

$$48 \cdot 50 \cdot 20 + \underbrace{48 \cdot 50 \cdot 2 + 48 \cdot 350}_{\text{additional runs for large systems}} = 69,600$$

optimization runs.

For each system, the deviation from the best found solution was calculated for each combination of neighbor function and iteration number. Note that the best found solution was calculated using the results from all three neighbor functions. Furthermore, the degree of freedom was calculated for each system. The degree of freedom quantifies the number of possible placements and is defined as n^s, where n is the number of nodes in the network and s is the number of freely placeable services (the logic services in our example). The deviations were grouped by the degree of freedom and averaged. Figure 2.29 shows the average deviation for the different combinations of neighbor functions and number of iterations. Please note the logarithmic scale of the x-axis. As expected, a higher number of iterations leads to better solutions. The measurements also show that neighborhood function 3 (using nodes at a two hop distance) shown in Figure 2.29(c) outperforms the other functions if a high enough number of iterations is chosen. For very low iteration numbers, neighbor function 1 is the best choice.

Neighbor function 1 performs best with very low iteration numbers because the diameter of the search space is smaller compared to neighbor functions 2 and 3. The diameter is the minimum number of iterations required to reach any other solution from a given solution. When services are moved randomly, every placement can be generated in at most s iterations, where s is the number of freely placeable services. The diameter for the other neighbor functons is a multiple of this value, because moving a service over a distance of n hops in the network requires n iterations, or $^n/_2$ respectively. If a very low number of iterations is chosen, a low diameter is beneficial because it minimizes the number of steps required to reach the optimum. If a suitably high number of iterations is selected, neighbor funtions 2 and 3 outperform function 1. The locality exploited by these functions increases the convergence towards the optimal solution at low temperature values and leads to a better overall performance.

2.6.3.3 Calculation Rule for the Limit l

In a practical setting, an user of the SA algorithm is faced with the question which value to chose for the limit l. From the measurements shown in Figure 2.29, we derived a function that yields a suitable value for the limit l for a given degree of freedom. This was done by selecting the lowest number of iterations that resulted in a deviation of less than 1% for each degree of freedom. Using a curve fitting algorithm, an logarithmic function

$$f_l(d) ::= max\{50, 14 \cdot log(d)\}$$

can be derived, which specifies the limit l as function of the degree of freedom d. The lower bound of 50 was used to enforce a minimum number of iterations for small scale networks. With f_l and the value 0.2 for λ, the optimizer in the ϵSOA platform can automatically adjust the configuration of the SA algorithm for a given system model. This ensures that enough iterations are performed to actually find a good solution in large networks and that no unnecessary iterations are performed for small networks.

2.6.3.4 Summary

In this section, we gave an in-depth analysis of the Simulated Annealing optimization algorithm used in the ϵSOA platform. We compared different neighborhood functions and parameter sets on a broad set of networks with varying complexity. Based on these results, we derived a calculation rule and a set of default parameter that allows to fully automatically tune the SA algorithm for a given embedded network in order to minimize the execution time of the optimization algorithm.

2.6.4 Optimization of System Availability

Besides the performance optimizations presented in the previous section, the service placement can also be used to optimize the system availability. By placing mission critical services on the most reliable nodes, the availability of applications can be increased. Using this technique, the achievable availability still depends on the availability of the nodes executing these services. To further increase the availability of applications in an embedded network, the ϵSOA platform allows exploiting redundantly available hardware. We already outlined in Section 2.5 how an application with redundant service instances can can be modelled during application design. Based on this information, the system can automatically react to the failure of individual components by replacing the failed component with one of the specified redundant component. To fully exploit the benefits of such an approach, redundancy also has to be considered during service placement. An example is that -for obvious reasons- it should be avoided to place a service instance on the same node as the redundant backup instance.

The ϵSOA platform supports an availability metric that allows optimizing systems w.r.t. to the availaibility of applications. At the current state, the metric allows to optimize systems based on hardware failures. However, the mechanisms presented in this section can be extended to incorporate software failures, too. As already mentioned in Section 2.5, the failure compensation mechanisms in the ϵSOA platform are optimized for single node failures which do not result in network partitions. As a consequence, we will present a metric that allows an optimization of the system availability under this boundary condition. The presented concepts can be easiliy extended to include network partitions. In this case, the reachability between service instances has to be checked in addition.

A prerequisite for optimizing the availability of an embedded network is knowledge

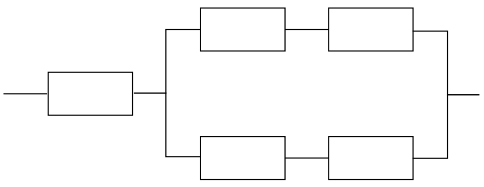


Figure 2.30: Example System Containing Series and Parallel Parts

about the individual components used in the network, i.e., the availability of the nodes. If accurate availability rates for individual nodes are available - e.g. through monitoring - the presented metric can calculate the overall system availability. If such detailed information is not available, the metric may also be supplied with reliability classes that quantify the expected availability and allow defining "risk" classes for nodes, e.g., battery powered nodes, wireless nodes, mobile nodes, etc.

We will first present the mathematical background for calculating system availability. After that we will explain the availability metric based on an example from a building automation scenario.

2.6.4.1 System Availability Basics

The availability of a system consisting of multiple independent parts can be calculated by modelling the system as a combination of parts, see Shooman[138]. Two combination types are possible: series of parts and parallel parts. In a series of parts, the combination fails if one part fails. Using parallel parts, the combination fails if all parts fail. These two basic types can be combined to create arbitrary complex composition. Figure 2.30 shows a combination consisting of an outer sequence. The second part of the outer sequence is a parallel combination of two parts, which in turn are a sequence of two parts.

For a series of n independent parts $A_1, \dots, A_n$, the overall system availability $A^s_{1..n}$ is defined as

$$A^s_{1..n} = A_1 \cdot A_2 \cdot \ldots \cdot A_n$$

By exploiting the associativity of multiplications, this can be reformulated to

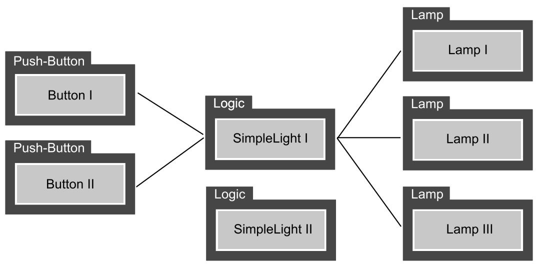
$$A^s_{1..n} = A^s_{1..n-1} \cdot A_n$$

For a parallel combination of n independent parts $A_1, \dots, A_n$, the overall system availability $A^p_{1..n}$ is defined as

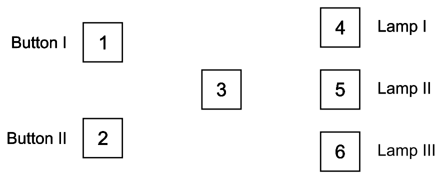
$$A^p_{1..n} = 1 - (1 - A_1) \cdot (1 - A_2) \cdot \ldots \cdot (1 - A_n)$$

Analogously to the serial combination of parts, this can be reformulated to

$$A^p_{1..n} = 1 - (1 - A^p_{1..n-1}) \cdot (1 - A_n)$$



(a) Application



(b) Embedded Network

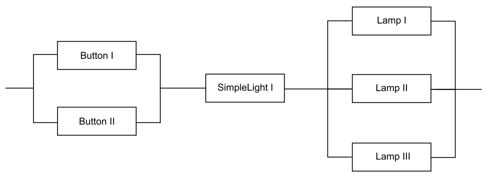
Figure 2.31: Example Scenario For Availability Metric

Proof:

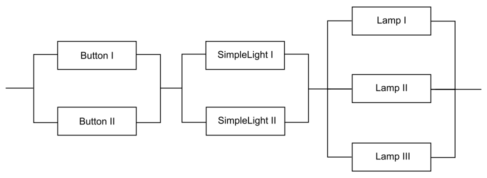
$$
\begin{aligned}
A^p_{1..n} &= 1-(1-A^p_{1..n-1})\cdot(1-A_n)\\
A^p_{1..n} &= 1-(1-\underbrace{(1-(1-A_1)\cdot(1-A_2)\cdot\ldots\cdot(1-A_{n-1}))}_{A^p_{1..n-1}})\cdot(1-A_n)\\
A^p_{1..n} &= 1-((1-A_n)-(1-A_n)\cdot(1-(1-A_1)\cdot(1-A_2)\cdot\ldots\cdot(1-A_{n-1})))\\
A^p_{1..n} &= 1-((1-A_n)-((1-A_n)-(1-A_1)\cdot(1-A_2)\cdot\ldots\cdot(1-A_{n-1})\cdot(1-A_n)))\\
A^p_{1..n} &= 1-(1-A_n)+(1-A_n)-(1-A_1)\cdot(1-A_2)\cdot\ldots\cdot(1-A_{n-1})\cdot(1-A_n)\\
A^p_{1..n} &= 1-(1-A_1)\cdot(1-A_2)\cdot\ldots\cdot(1-A_{n-1})\cdot(1-A_n)\\
A^p_{1..n} &= A^p_{1..n}
\end{aligned}
$$

2.6.4.2 Availability Metric

We will describe the availability metric based on an example scenario shown in Figure 2.31. Assume we want to place the application shown in Figure 2.31(a) on the embedded network illustrated in Figure 2.31(b). The taks of the application is to provide a simple lighting application in a building automation scenario. It comprises two push buttons, Button I and Button II that can be used to toggle the lights in the room on or off. The room contains three lights, Lamp I, Lamp II and Lamp III. The application logic is contained in the SimpleLight I service. Ignore the SimpleLight II service for the moment. Note that the only service that can be freely placed in the embedded network is the SimpleLight I service. All other services depend on the availability of specific hardware devices and are therefore fixed to a specific node. Figure 2.31(b) shows these assignments: the buttons are attached to the leftmost nodes, the lights to the rightmost nodes. The central node possesses no sensor or actuator devices. In order to keep the example simple, assume that all nodes communicate via wireless links and are in the radio range of each other (what is very likely if they are placed in the same room). The network therefore possesses a full mesh communication infrastructure and each node can directly communicate



(a) Basic Scenario



(b) Scenario with Replicated Logic

Figure 2.32: Availability View

with each other. Assume that the buttons are battery powered, perhaps even mobile. As a consequence the availability of the buttons is lower that the availability of the remaining nodes. Assume the buttons are available with 90% and all other nodes with 95%[17].

The lighting application will work als long as there is at least one button and one lamp available. From an availability point of view, the buttons are operating in parallel. The same holds for the lights. The overall application is a sequence of the button group, the logic service and the lamps, as depicted in Figure 2.32(a). Assume the logic service SimpleLight I is installed on node 3. In this case, the availability of the whole application can be easily calculated based on the formulas presented in the previous section. The availability of the button group is:

$$1 - (1 - 0.9) \cdot (1 - 0.9) = 0.99$$

The availability of the group of lamps is:

$$1 - (1 - 0.95) \cdot (1 - 0.95) \cdot (1 - 0.95) = 0.999875$$

The overall availability of the system therefore is:

$$\underbrace{(1 - (1 - 0.9) \cdot (1 - 0.9))}_{\text{Buttons}} \cdot \underbrace{0.95}_{\text{Logic}} \cdot \underbrace{(1 - (1 - 0.95) \cdot (1 - 0.95) \cdot (1 - 0.95))}_{\text{Lamps}} = 0.9403824375$$

This calculation method is not applicable in general. Assume the logic service is installed on one of the lamp nodes, e.g., node 4. In this case, the availability of the individual services is not independent anymore. Everytime the logic service is up, we know for sure that also Lamp 1 is up (assuming the lamp itself does not fail). The formulas presented in the previous section require independent components and therefore cannot be applied anymore. To calculate the system availability, we have to distinguish two scenarios:

[17]Real world components will typically have higher availabilities, these numbers were chosen to get a concise example.

Scenario 1: Node 4 is available In this case, both the logic service and at least one lamp are available. As a consequence, the availability is purely based on the availability of the switches, which we already calculated. The availability in Scenario 1 therefore is:

$$A_{S1} = 1 - (1 - 0.9) \cdot (1 - 0.9) = 0.99$$

Scenario 2: Node 4 is down In this case, the logic service is not available. Because it is a required part of the application, the overall application is not available, too. The availability in Scenario 2 therefore is $A_{S2} = 0$.

The likelihood for Scenario 1 in a given time interval is equal to the availability of node 4, which is 95%. With this, the overall availability can be calculated as

$$0.95 \cdot \underbrace{(1 - (1 - 0.9) \cdot (1 - 0.9))}_{A_{S1}} \cdot 0.05 \cdot \underbrace{0}_{A_{S2}} = 0.9405$$

The availability in this case is a litte bit higher than the availability in the previous example. In both examples, the system will fail when the node executing the logic service fails. In the first example we need at least two additional nodes for the application, one switch and one light. In the second example, we only need one additional node, one switch. As a consequence, the application in the second example has a little higher availability, because it requires only two available nodes, whereas the application from example one requires three.

The vailability in both of the example presented in this section was clearly dominated by the availability of the logic service. Assume the user wants to remove this single point of failure and install a replica of the logic service. This can be easily done during application development by creating a second instance of the logic service, (in this case LogicService II) and assigning it as redundant backup to LogicService I. The resulting service chain from an availability point of view is shown in Figure 2.32(b).

Assume that LightLogic I is placed on node 4, just like in the second example. Furthermore assume that the replacated logic should be installed on node 5. In this case we have two dependencies between services: one between LogicService I and Lamp I and the other between LogicService II and Lamp II. The overall availability can be calculated just like in the previous example by splitting the calculation into multiple scenarios:

Scenario 1: Node 4 and Node 5 are available If both nodes are available, at least two lamps are available, too. As a consequence the application availability is determined by the availability of the buttons.

$$A_{S1} = 1 - (1 - 0.9) \cdot (1 - 0.9) = 0.99$$

Scenario 2: Node 4 is available, Node 5 is down Just like in Scenario 1, the availability of the system is determined by the availability of the buttons, because at least one lamp - Lamp 1 - is always available.

$$A_{S2} = 1 - (1 - 0.9) \cdot (1 - 0.9) = 0.99$$

Scenario 3: Node 4 is down, Node 5 is available This scenario is similar to Scenario 2.

$$A_{S3} = 1 - (1 - 0.9) \cdot (1 - 0.9) = 0.99$$

Scenario 3: Node 4 and Node 5 are down If none of the replicated logic services is avaialbe, the whole application is unavailable.

$$A_{S4} = 0$$

The resulting application availability can be calcuated based on the probabilities of the individual scenarios which is:

$$\left(\underbrace{0.95 \cdot 0.95}_{\text{Scenario 1}} + \underbrace{0.95 \cdot 0.05}_{\text{Scenario 2}} + \underbrace{0.05 \cdot 0.95}_{\text{Scenario 3}}\right) \cdot \left(\underbrace{1 - (1 - 0.9) \cdot (1 - 0.9)}_{\text{Buttons}}\right) = 0.987525$$

2.6.4.3 Calculation Scheme for Availability Metric

This calculation scheme can be generalized. The algorithm distinguishes between service instances with availability dependencies, i.e., instances that are placed on the same node, and independent instances. To speed up the calculation, independent instances are aggregated by using the formulas presented in the beginning of this section. If a parallel group contains multiple independent instances, these are aggreagated. The same holds for a sequence of independent instances.

After this simplification step, the availability for the different combination scenarios of dependent instances are iteratively computed by the algorithm. In a final step, these individual availabilities are aggregated to an overall availability using the probability of each combination as weighting function, like in the example. The total number of scenarios is 2^k, where k is the number of nodes with non-independent services. In typical automation use cases, this number is fairly small because service compositions have a limited size and many of the involved services are placed on dedicated sensor and actuator devices and are therefore independent.

The calculated availability metric for a placement can be used as stand-alone optimization criterion or combined with the other metrics presented in this section.

2.6.5 Ongoing Work: Service Placement Optimization with Genetic Algorithms

As mentioned in the beginning of this section, we performed a series of initial test to compare the results of a Genetic Algorithm to the results created with Simulated

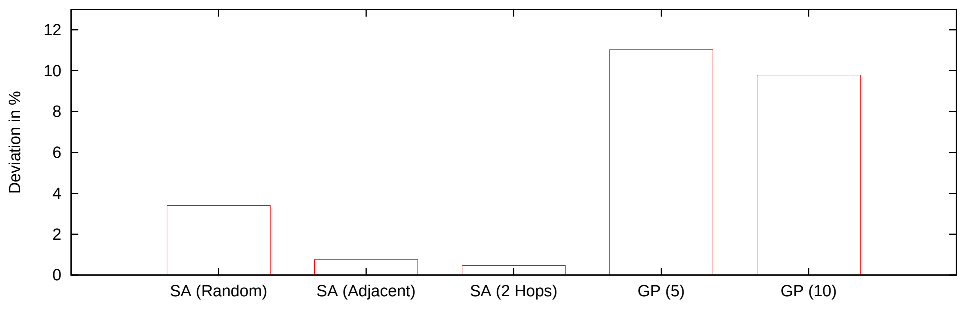


Figure 2.33: Comparison of Different Optimization Algorithms for Scenario 20

Annealing. For smaller scenarios, both algorithms are able to determine the optimal solution. For larger scenarios, differences are observable. Figure 2.33 shows the averaged deviation from the optimum for systems based on Scenario 20. A total number of 500 systems was generated. Each system was optimized using the Simulated Annealing Algorithm, with the three different neighbor functions and a number of 800 iterations, and the Genetic Pogramming Algorithm with a gene pool of size 5 and a value of 5 and 10 mutations per generation. The number of generations used in the Genetic Algorithm was 160 for a pool of size 5 and 80 for a pool of size 10. The most resource intensive operation in both algorihms is the calculation of the evaluation metric for each tested placement. The settings mentioned above ensure both algorithms use the same number of metric evaluations, this is 800. As Figure 2.33 shows, the Genetic Algorithm has a much higher deviation compared to the Simulated Annealing algorithm, i.e., often returns a worse placement. The reason for this is a well known limitation of Genetic Algorithms. While Genetic Algorithms tend to yield solutions that are close to the optimum, they often require many generations for finding the last few mutations that result in an optimal solution. If the number of generations is increased the results returned by the Genetic Algorithm are getting better, but this comes at a considerable increase of computation costs. We confirmed these results for other combinations of gene pool sizes and mutations per generation. In all cases, the average deviation was considerably higher compared to Simulated Annealing. Due to this reason, the current implementation of the ϵSOA platform uses a Simulated Annealing algorithm for optimizing the placement of services.

A solution that often improves the results of Genetic Algorithms is a combination with greedy heuristics, such as hill climbing. In this scenario, the Genetic Algorithm ensures that the optimization does not get stuck in a local optimum, whereas the heuristic ensures a fast convergence to the actual optimum. We are currently investigating how such a heuristic could look like and whether it can be used to tune the performance of our Genetic Algorithm. We are also investigating whether a crossover function can improve the results created by the Genetic Algorithm. A possible crossover function is to exploit the presence of multiple applications. In this case, a new genome is created by mixing the placements for individual applications.

In order to create a set of pareto optimal solutions w.r.t. multiple metrics, the

Genetic Algorithm can be extended with a special sorting function. Deb et al.[25] propose such an algorithm called NSGA-II (Non-dominated Sorting Genetic Algorithm). We are investigating how this approach can be integrated into the GA for service placement and the development tools in the ϵSOA platform.

2.6.6 Related Work

Wittenburg et al.[169] present an overview of research activities considering service placement in ad-hoc networks. This work shares some properties with the service placement problem analyzed in this section, however there are some important differences. The special characteristics encountered in control oriented networks, especially the longetivity of applications and the stable network structure, require different optimization approaches compared to ad-hoc networks with more dynamics. A dedicated optimization/configuration phase as used in the ϵSOA platform is possible due to this stability and also needed to ensure the optimality of the created solution which may be executed based on the derived placement for weeks, months or even years.

Yick et al.[171] study the problem of distributung data logger services in a randomly organized sensor network. The goal is to achieve a complete coverage of the observiced area, to provide load balancing between nodes and to a minimize of the energy consumption for gathering and transmitting sensor readings. The latter can be achieved by minimizing the number of data loggers and optimizing their placement in the network. The basic problem tackled by Yick et al. is comparable to the service placement problem presented here, nevertheless there are some domain specific differences. A problem challenge encountered in the work of Yick et al. is that the communication between nodes is influenced by obstacles in the environment, which requires a line of sight analysis when calculating communication paths. Another difference is that Yick et al. optimize the placement for a single application (i.e. try to minimize the resource utilization), whereas control oriented networks have to be designed to support the simultaneous execution of multiple applications, which compete for resources (i.e. also optimize the assignment of resources to different applications).

In TinyDB[93, 94], sensor data can be acquired and transformed using a SQL-like syntax. The execution of these queries is based on a tree oriented network structure. The placement of the individual operators of these queries, such as selections and joins, are placed according to a heuristic. This approach is feasible for sensing oriented networks with a dedicated sink. It cannot be applied to control oriented networks where multiple applications with differing communication behavior compete for the available resources.

Besides this sensor network focused work, optimization techniques/challenges comparable to the ones described in this section can also be found in the IT domain.

Data Stream Management Systems An optimization technique in Data Stream Management Systems (DSMS) that can reduce the transmitted data volume considerably is *stream sharing*, see Kuntschke[84] for more information and a detailed list of publications in this research area. The idea of stream sharing is to detect commonalities in the data usage of different queries and to exploit these commonalities by only transmitting the shared data once. The sharing of measurements presented in this chapter can be seen as a variant of such optimizations. An interesting direction for future research is an analysis whether stream sharing is applicable and actually beneficial in control oriented embedded networks. Our experience from prototypical implementation indicates that application often either require the exactly same data or have very little commonalities. The former is often the case for "raw" sensor data, which is consumed by multiple sinks. In this case, a multicast communication achieves the same results as a stream sharing approach. In contrast to DSMS, the consumers of this data typically apply complex processing tasks and the resulting output is so specialized that it cannot be reused in multiple applications. Examples for such transformations are control loops or other application logic. A mere filtering or aggregation of data is rather uncommon. The possibilities for stream sharing are therefore limitied. Another difficulty is that data processing tasks in control applications do not necessarily have a clearly defined semantic (as operators in a DSMS typically have). This complicates the detection of commonalities as the exact characteristics of the produced data streams can be unclear. Nevertheless, there surely are situations where stream sharing can be beneficial, especially in monitoring/data analysis heavy networks.

Data Center Optimization Virtualization and Software as a Service techniques promise cost savings for the operaters of data centers. The possibility to move enterprise services between hosts without downtime and to scale the number of service instances according to the current utilization can be used to increase the individual utilization of servers in a data center and consequently allows reducing the overall number of required servers. Gmach[46] shows that a colocation of services with complementary resource demands, e.g., daytime vs. nighttime activity, and a proactive scaling of applications based on load estimations can reduce the number of hosts while maintaining the same QoS. The optimization problem of placing services in a data center is related to the service placement problem encountered in embedded networks. Both aim at optimizing the resource utilization in the system, however with different goals and boundary conditions. In data centers, the main goal is to maximize the utilization (by keeping the QoS at the same level) in order to support more applications/customers with the same hardware. In embedded networks, the optimization goal typically is to achieve a certain QoS goal or to homogenize the resource utilization. The reason is that nodes often have additional tasks besides the mere processing of data, e.g., attached sensors or actuators or the forwarding of data in a multi-hop network, and cannot be removed, even if it is possible to move all processing tasks to other nodes. The goal therefore is not to minimize the number of required nodes, but to optimize the exploitation of available resources to maximize QoS, battery lifetime, etc.

Virtual Machine Grid Computing An optimization problem comparable to the service placement problem presented in this work is studied by Sundararaj et al.[150]. The authors present a virtual network layer for grid computing based on virtual machines. In such a network, the placement of virtual machines can be optimized w.r.t. network and utilization metrics. The boundary conditions imposed by a grid computing environment differ from the boundary conditions imposed by control oriented networks. Nevertheless there are similarities which could be an interesting starting point for a more detailed comparison of both approaches.

2.6.7 Summary

In this section we presented optimization techniques used in the ϵSOA platform to optimize the execution of applications in heterogeneous embedded networks. We showed how the acquisition of measurements can be optimized when sensors are used in multiple applications and how the placement of services can be optimized based on communication characteristics and the characeristics of the underlying network infrastructure. The current implementation uses Simulated Annealing to solve the service placement optimization problem. In ongoing work , we are examining a Genetic Algorithm which could be used as an alternative approach for solving the optimization problem. Another direction for optimization, which we are investigating in ongoing work, is the optimization of schedules for TDMA based communication media, such as FlexRay[18]. The challenge here is to derive a schedule which guarantees that all end-to-end latency requirements imposed by the running applications are met.

[18]`http://www.flexray.com/`

CHAPTER 3

Runtime Environment

In the previous chapter, we presented the design principles of the ϵSOA platform and described an application development workflow based on service oriented principles. In order to actually execute the modeled services and service compositions, the nodes in the embedded network have to provide a corresponding execution environment. This functionality is offered by the ϵSOA Runtime Environment, which is installed on every node in the embedded network. We will first present the overall architecture of the ϵSOA Runtime Environment in Section 3.1, followed by a detailed description of the individual components. The ϵSOA platform uses XML to transmit data between nodes and to store information on the nodes in the embedded network. In Section 3.2 we will analyze different XML encoding formats and show that binary XML techniques can meet the resource constraints imposed by microcontrollers.

In many application fields, it is desirable to use self descriptive nodes, i.e., nodes that contain not only the service implementation, but also all information required for the configuration and integration of the installed services. If a self descriptive node is attached to a network its capabilities and characteristics can be retrieved and added to the system model. This reduces development overheads and allows a synchronization of the model used in the planning and development tools and the actual system. In Sections 3.3, 3.4 and 3.5 we will present XML based description languages that allow storing the relevant parts of the system model directly on the node.

In Section 3.6 we will present the communication module of the ϵSOA middleware. The ϵSOA platform uses a binary XML based message format. In order to achieve a high parsing performance, a code generator is used to create tailored message parsers and databinding code. This code generator is presented in Section 3.7.

The management of large scale embedded networks requires a management and configuration protocol. We will present such a protocol in Section 3.8. This protocol can also be used to facilitate a cross layer information exchange, i.e., to pass information from the network stack up to the application level and push information down the network stack.

The flexibility of embedded networks can be increased if services can be added and removed at runtime. In Section 3.9 we will present such a mechanism and a workaround based on service libraries, which can be used if the underlying operating system does not support the dynamic loading of code.

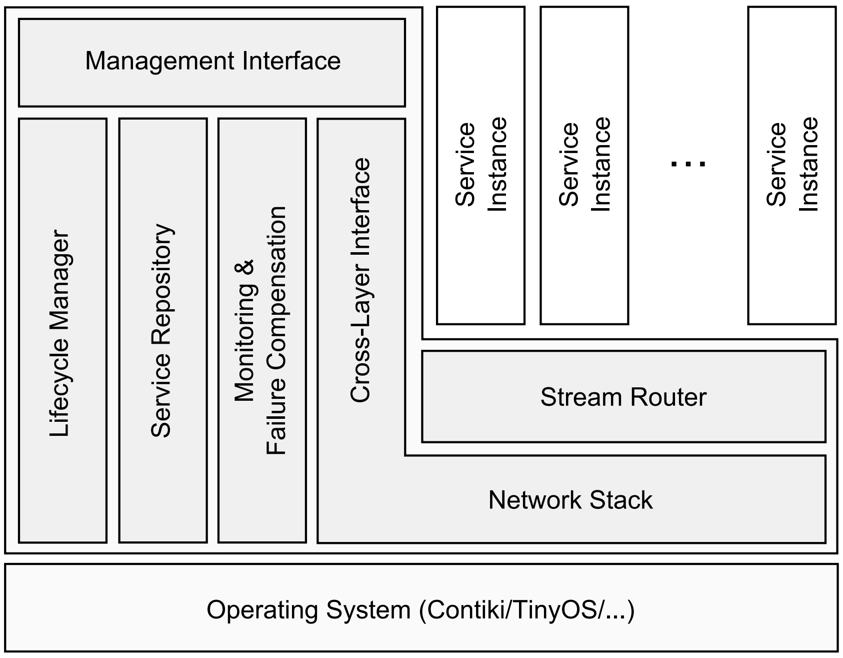


Figure 3.1: The ϵSOA Runtime Environment - Architectural Overview

3.1 Runtime Environment Architecture

The ϵSOA Runtime Environment is installed on every node in the embedded network. It provides a service execution environment and basic functionality that can be accessed by each service running on the node. We will present a short overview of the purpose and functionality of each runtime component here. A detailed description can be found in the corresponding sections of Chapter 3 and Chapter 4.

3.1.1 Runtime Components

Figure 3.1 shows the architecture of the ϵSOA Runtime Environment. It comprises seven components:

Network Stack The ϵSOA network stack sits on top of the transport/routing layer protocols provided by the underlying operating system. It provides a basic communication interface and handles the transmission of messages across network boundaries in heterogeneous embedded networks.

Stream Router The Stream Router augments the Network Stack with functionality for handling data streams. The Stream Router specifies where produced data is sent. It stores non-functional requirements (e.g. reliability) for each stream and handles the splitting of data streams if a data stream is consumed by multiple services. The ϵSOA network stack and the Stream Router are presented in detail in Section 3.6.

Cross-Layer Interface The Cross-Layer Interface can be used to perform a message exchange between the communication layer and the application layer. The application layer gets access to topology information and communication characteristics through the Cross-Layer Interface, whereas the Communication Layer can retrieve application layer information, e.g. data rate estimations for data streams. The exchanged information can be used to optimize the configuration in the corresponding layer and is used as input for the optimization algorithms presented in Section 2.6. The Cross-Layer information exchange is explained in Section 3.8.

Monitoring & Failure Compensation The ϵSOA platform offers mechanisms for compensating the failure of nodes in the embedded network. The monitoring of neighboring nodes and the execution of compensatory actions is performed by the Monitoring & Failure Compensation module of the ϵSOA platform. The supported failure compensation mechanisms are explained in detail in Section 4.1.

Lifecycle Manager The Lifecycle Manager handles the instantiation of services and the management of service instance states, i.e., the starting, stopping and removal of service instances. It implements the Service Lifecycle Model presented in Section 2.4. The Lifecycle Interface uses the Service Repository to gain access to the executable code and metadata description of services.

Service Repository The Service Repository contains the executable code and the metadata description of each service. Services can be added to the Service Repository during the initial programming of the node, or added at runtime if the underlying operating system supports the dynamic loading of code. Both mechanisms are explained in Section 3.9.

Management Interface The configuration of module and service parameters can be performed through a Management Interface. The XML based Management Interface used in the ϵSOA platform is explained in Section 3.8.

Service Instances The components mentioned above provide an execution environment that can be used to execute and manage an arbitrary number of different Service Instances, only limited by the available system resources (memory, CPU power, ...) on a node.

3.1.2 Summary

In this section, we described the main building blocks of the ϵSOA Runtime Environment. It provides the execution environment for services on each node. The Runtime Environment is created and configured based on the requirements specified in the System Model and installed on each node in the network. This initial installation can be pre-configured with services and service instances to build ready-to-use nodes (e.g. to design sensor/actuator devices that are shipped with corresponding hardware services) or empty nodes that initially contain no services and act as generic service hosts. In both cases, the node configuration can be changed and new services and service instances can be deployed at runtime in order to adapt the node to the requirements of a specific installation. The configuration of the Runtime Environment and the installation and management of services and service instances can be performed through a management interface, which is used by the development and administration tools described in Chapter 5.

3.2 Encoding Techniques

The resource constraints encountered in embedded networks necessitate efficient encoding techniques for storage, transmission and processing of messages. In most cases, the most power consuming device on a wireless node is the radio module. A compact message format reduces the overall power consumption, what is especially important for battery powered nodes. A compact data representation is also beneficial from a utilization point of view. Smaller message sizes allow higher data rates on links with limited bandwidth. Nodes in an embedded network often possess comparably weak CPUs and only a very limited amount of RAM for storing intermediate results when parsing a message. The encoding scheme must be simple enough to support an efficient implementation on such systems.

In this section, we will analyze different encoding techniques w.r.t. their suitability for embedded networks. An efficient, yet flexible data format is not only required during message transmission, but also for storing information on the nodes in an embedded network, such as service descriptions, configuration information, service compositions, etc. The requirements are identical in both application fields and the presented encoding techniques can also be used to store such information efficiently on resource constrained devices.

Two factors contribute to the size of the data representation: the data format overhead and the encoding of data types. The data format overhead quantifies the additional information used to encode data types and other semantic information. In an embedded network, most of the data transmissions are periodic and known a priori. This can be exploited by using a transmission format that contains solely the data payload and that uses the interface description of the service on the client and the server side to interpret the data correctly. This technique can greatly reduce message sizes for periodic transmissions and should be supported by the data format.

The encoding of data types is the second key factor that influences the message size. In many cases, services in the embedded network will exchange numeric data which has to be encoded efficiently. When communicating with external Web services, textual data has to be transmitted, too. Through an efficient encoding of numeric and textual data, the size of these messages can be reduced by a large amount.

Besides the compact data representation, the message format should also be extensible and flexible enough to support complex, structured data types. Not all sensor services will consume simple numeric values; consider for example a RFID reader which will create tuples comprising the EPC code of the observed item, the timestamp of detection and the id of the reader.

Even more complex structures will be encountered in scenarios where an interaction between the embedded network and enterprise Web services occurs, e.g. in a shop floor integration scenario. In these cases complex XML documents have to be transmitted and processed by the embedded network. This leads to another requirement: XML support. The data format has to provide a possibility to efficiently interact with XML based services. Because parsing XML documents requires large amounts of memory and CPU power it is not feasible to simply transport the document to a service in the embedded network. This issue is solved by using message converters that intercept incoming XML messages and transform them to the data format used in the embedded network. The returned outputs are transformed back to XML again. The message format should provide a way of mapping XML documents to the message format and vice versa. It is decisive that this mapping is applicable to any kind of XML message and can be done by a generic gateway. The development overhead for designing a manual mapping between the external XML document and the data representation used inside the network should be avoided.

Summing up a message format should have the following properties:

- low format overhead
- low encoding overhead
- high flexibility and extensibility
- support for XML data

Looking at these properties, XML itself is a possible candidate for the message format. It provides the required flexibility and inherently supports nested XML data. The major drawback of plain XML is the very large overhead due to the verbose markup and the string based encoding of data types. This overhead is not only problematic in the domain of embedded networks, but also in the Web service domain. Processing large numbers of XML messages (such as SOAP[165] calls) will challenge even modern servers. As a consequence, a lot of effort was done in the past years to improve the processing speed of XML parsers and to reduce the message size transmitted over the network. Because the XML representations generated by these approaches are not human readable anymore, they are commonly referred to as *binary XML* technologies. In the following paragraphs we will analyze these binary XML encoding techniques and ASN1.0 PER, a popular binary encoding format, w.r.t their suitability as encoding format for embedded networks.

To keep the discussion concise, we will focus on the properties mentioned above and defer the discussion regarding the parsing complexity to Section 3.7, where we will show how efficient message parser for embedded devices can be realized.

Abstract Syntax Notation One (ASN.1) A popular encoding standard in the IT domain is the Abstract Syntax Notation 1.0 (ASN.1)[69], which is standardized by the International Telecommunication Union (ITU). ASN.1 itself is not an encoding format, but a description language for data formats. Based on this abstract data definition, different encoding formats can be applied to represent the data. The most compact data format are the ASN.1 Packed Encoding Rules (PER)[70]. PER defines an efficient binary format with very little overhead and therefore is an interesting encoding technique for embedded networks. It provides the required flexibility by supporting a broad range of data types and the possibility to create complex data structures out of these types. Based on the syntax definition of an interface it is also possible to generate an efficient parser for messages directed at this interface. ASN.1 therefore fulfills the first three requirements mentioned above.

It is possible to represent the information contained in a XML document in ASN.1. The de-facto standard for data definitions in the Web service world is XML Schema[162, 163] (or RelaxNG[108] to a lesser degree). In order to provide an automatic conversion between messages in both domains, a mapping between XML Schema and ASN.1 is required. The ITU specifies such a mapping between a XML Schema definition, and a ASN.1 syntax definition[71]. The opposite direction is possible, too. By using these mappings, it is possible to design a message converter that can convert ASN.1 PER documents to XML document and back again. However there are some limitations due to information loss during the conversion. A big difference between ASN.1 and XML Schema is the expressiveness w.r.t to data types. XML Schema offers the possibility to define data types on a fine grained level using facets. With these facets it is possible to define data ranges for numeric values, declare enumerations or restrict data types to specific patterns defined by regular expressions. Many of these features cannot be represented in ASN.1 very well. It is possible to mimic these facets via user-defined constraints to an ASN.1 definition, but these are mere annotations and will be ignored by ASN.1 parsers. On the other hand, many of these restrictions have very little or no influence on the data representation used when encoding messages. The additional information lost during conversion might therefore be compensatable by using additional type checking in the application logic.

Summing up, ASN.1 PER provides an efficient and compact binary representation of data values. It also supports the representation of XML based data - but there are some losses during the conversion of messages.

Compressed XML A straightforward approach for reducing the size of an XML document is the application of data compression algorithms. Because the markup (and sometimes also the payload) contains recurring occurrences of the same tokens, such as tag names, this can result in a considerable reduction of the document size. The drawback of this approach is that it requires a fairly high amount of processing power on the nodes, because compression algorithms are CPU intensive. Another drawback is a high memory demand. After decompression, the resulting XML document will occupy a considerable amount of RAM on the node. We are using the gzip[58] compression algorithm for our analysis, because it is available on a wide variety of platforms and it is patent-free. There are a lot of other compression algorithms besides gzip, such as bzip2[1] for example, which might be interesting for compressing XML data. The results of our experiments with gzip showed that compressed XML is still too large for embedded networks. We therefore did not analyze other compression algorithms in detail.

VTD-XML An approach that aims at speeding up the processing of XML documents is VTD-XML[170]. This is done by adding Virtual Token Descriptors (VTDs) to XML documents. A VTD encodes the offset, length, token type and nesting depth of a token in an XML document. Navigation in the XML document can be performed based on these tokens what results in a speedup compared to traditional parsing mechanisms. Because VTD-XML is a parsing technique, the message size cannot be reduced for the transmission over the network. Parsing messages with VTD-XML even requires additional memory for storing the token descriptors. Its applicability in the domain of embedded networks is very limited, because the memory available on every node is very scarce. We will therefore not analyze VTD-XML in more detail here and concentrate on other encoding techniques that allow reducing the size of transmitted XML messages.

WBXML The Binary XML Content Format Specification (WBXML)[116] has been developed to ease the handling of XML based messages on mobile phones. It aims at reducing the transmission size of XML based documents and improving the parsing performance of XML documents. WBXML encodes the parsed XML document, i.e., the structure and content of the encountered XML entities with a binary format. Any meta information contained in the original XML document is removed during the conversion. WBXML does not use schema information for encoding documents. As a consequence documents are always self-contained, i.e., can be reconstructed without external knowledge.

XML Fast Infoset (FI) A technique that aims at combining the goals of reduced document size and increased parsing performance is Fast Infoset (FI)[72]. The idea of FI is to use an efficient binary representation for the XML data model. FI relies on the Abstract Syntax Notation 1.0 (ASN.1) to (de-)serialize XML documents. Note that there is a fundamental difference between XML FI and the XML to ASN.1 mapping described in the first paragraph of this section. XML FI encodes an XML tree with ASN.1, i.e., there is an ASN.1 construct for storing a tag, an attribute, a namespace, etc. This encoding can be done without any XML Schema information about the XML document. The XML to ASN.1 mapping on the other hand does not encode the XML document itself, but only the *information* contained in the XML document. The resulting document is not a XML tree anymore and cannot be parsed with a SAX or DOM parser. The benefit of XML FI is that there is no information loss involved if a document is encoded with XML FI. On the other hand, the resulting message size will typically be larger compared to an information centric approach, because the XML structure and especially the names of tags and attributes are still contained in the document.

An important building block of FI is a vocabulary that maps verbose strings to compact numeric representations. XML FI supports two different modes for storing this vocabulary. In the default mode, the vocabulary is contained in the encoded document. We will refer to this mode as FI. In the second mode, the dictionary is not added to the document but kept at an external location. This mode is interesting if many documents with similar structure should be encoded. In this case, the vocabulary has to be exchanged only once and does not have to be added to every single document. This mode results in smaller document sizes. We will use FI-Dict to refer to this mode. The vocabulary exchange can be done during the creation of a data stream in the embedded network. Every element of the data stream can then use this external vocabulary.

Efficient XML Interchange (EXI) The Efficient XML Interchange (EXI) format aims at combining the goals of high processing performance and compact message sizes. EXI is based on a grammar that enumerates the possible elements occurring at every position of an XML document. This, combined with an efficient binary representation of numeric data types, creates a very compact message format that is easy to parse at the same time. EXI supports two modes for encoding a XML document: by using a built-in grammar or by using a schema-informed grammar. A built-in grammar is contained directly in the EXI encoded XML document. The resulting document is self-contained, i.e., an EXI decoder can reconstruct the XML document without any further information. We will call this encoding technique EXI. The embedded grammar also has a drawback: it has to be added to every single XML document, even if a sequence of documents referring to the same XML Schema should be encoded. This redundancy can be avoided by using schema-informed grammars. In this case, a grammar is created once for each XML Schema definition. This grammar encodes all valid XML documents that can be created based on this schema. We will call this approach EXI schema-informed (EXI-SI). Because the schema is fixed, it is sufficient to create and exchange this grammar once. It can then be used to en-/decode all documents corresponding to this schema. This results in smaller message sizes and can also be used to tune the performance of the EXI en-/decoder implementations. We will show in Section 3.7 how a tailored EXI en-/decoder can be generated.

EXI-SI is also capable of encoding documents that differ from the original XML Schema definitions. The deviations are encoded by using built-in grammar definitions that overwrite the schema-informed global definitions. This mechanism is very useful for encoding values that can have arbitrary values in theory, but use a limited subset of these values in a practical setting. A good example of such a value is a data definition used in a service description. Assume a user can define custom datatypes identified by a string value, or used built-in datatypes. In such a setting, the reference to a data type has to be encoded with a generic String literal because the names of the user defined types are not known beforehand. On the other hand, most of the data types will reference the built-in definitions, which are known beforehand. When using EXI, one can define an enumeration containing all the build-in datataypes. References to these types will be encoded very efficiently based on an index pointing to the corresponding value in the enumeration. If the reference targets a user defined datatype, which is not contained in the enumeration, EXI can encode this value anyway by overwriting the enumeration defined in the XML Schema and replacing it with a String based reference. This technique can be used in many situations and can yield a considerable reduction of document sizes if commonly used values for attributes and elements are known beforehand.

3.2.1 Test Cases

We defined a set of test cases that represent typical message types exchanged in embedded networks. The test cases cover a broad range of complexity, starting with documents containing only single measurement values, over documents containing more complex, compound data values, to XML documents encountered during communication with external Web services. The goal of these tests is to quantify the document size created by the different encoding techniques. In order to keep the examples concise, we use plain XML documents in this section to compare the performance of the different encoding strategies. In a practical setting, the messages exchanged in the embedded network will be a bit more complex, because the exchanged data values are embedded in a message format (just like messages exchanged by Web services are embedded in SOAP). This results in slightly bigger XML documents. Nevertheless, the results and conclusions presented in this section remain applicable in the more complex setting, too.

3.2.1.1 Single Data Value

Listing 3.1 shows an example document for a sensor delivering a single data value, e.g. a measurement stemming from a temperature sensor. The size of this data value has a, albeit small, influence on the resulting document size. We therefore used three instances of this document with data values of different sizes. The "short" variant uses a value of 1, as shown in the example, the "medium" variant a value of 120, and the "large" variant a value of 2.000.000.

```
<esoa:message xmlns:eSOA="http://www3.in.tum.de/eSOA"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www3.in.tum.de/eSOA single.xsd ">
  <data1>1</data1>
</esoa:message>
```

Listing 3.1: Message Containing a Single Data Value

3.2.1.2 Multiple Data Values

In some cases, a service delivers multiple data values at once, e.g., a weather station might supply a status report comprising a temperature and a humidity value in a single measurement. In this case, multiple data values are contained in the same message. Listing 3.2 shows an example document containing two data values. To quantify the influence of the data values on the document size, we used three different sizes for the elements, as we did in the previous example.

```
<esoa:message xmlns:eSOA="http://www3.in.tum.de/eSOA"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www3.in.tum.de/eSOA multiple.xsd ">
  <data1>1</data1>
  <data2>0</data2>
</esoa:message>
```

Listing 3.2: Message Containing Multiple Data Values

3.2.1.3 Compound Data Types

Not all data exchanged in an embedded network will be based on single numeric values, but will instead be based on compound data types composed of multiple simple data types. A possible example of such a message is shown in Listing 3.3. It shows a document containing a reading of a RFID reader, a so called RFID event. The RFID event is a triple comprising the Electronic Product Code (EPC) of the scanned item, the time the item was scanned and the identifier of the reader that scanned the item, which is an EPC again. An EPC, as specified by the EPCglobal consortium[35], is a unique identifier for an item. The exact information stored in the EPC depends on the encoding format used. Typical fields are an identifier for the manufacturer of the product, a product group identifier and a serial number. EPCs are stored as 96 bit integers, given in hexadecimal notation in the example.

```
<esoa:message xmlns:eSOA="http://www3.in.tum.de/eSOA"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www3.in.tum.de/eSOA complex.xsd ">
  <data1>
   <epc>1234567890AB1234567890AB</epc>
   <timestamp>2002-10-10T17:00:00Z</timestamp>
   <reader>FF1234567890FF1234567890</reader>
  </data1>
</esoa:message>
```

Listing 3.3: Message Containing a Complex Data Value

3.2.1.4 Collections

Another source of complex data types are collections. Listing 3.4 shows a document containing a RFID bulk. A RFID bulk is a group of elements that is bundled together, e.g., a palette of goods. To ease the further processing of RFID events, a RFID reader can aggregate the individual readings of such groups in a single message, resulting in a list of RFID events.

```
<esoa:message xmlns:eSOA="http://www3.in.tum.de/eSOA"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www3.in.tum.de/eSOA list.xsd ">
  <data1>
    <event>
```

```
      <epc>...</epc>
      <timestamp>...</timestamp>
      <reader>...</reader>
    </event>
    [...]
    <event>
      <epc>...</epc>
      <timestamp>...</timestamp>
      <reader>...</reader>
    </event>
  </data1>
</esoa:message>
```

Listing 3.4: Message Containing a List of Complex Data Values

3.2.1.5 Interaction with Web Services

Embedded networks do not operate in isolation but are more and more integrated into larger IT networks and exchange information with Web services located in these networks. It is not possible to process all XML documents exchanged between nowadays Web services on resource constrained devices. Some of these documents are so complex that the contained information is simply larger than the available RAM on a node in an embedded network. Often nodes in an embedded network do not require all information contained in such XML documents but only a subset. This subset can be extracted easily from the original message using XPath or XQuery processors.

In the long run, we will most likely see a growing number of Web services that offer documents that are small enough to be processed on embedded devices. Until now, there was little demand for Web service interfaces that produce small XML documents. Instead, service developers tried to stuff as much information as possible into the exchanged XML documents to support a broad variety of use cases. In most situations, a more fine grained access to the same information is sufficient to reduce the message complexity to a level that can be processed by embedded devices.

There already are Web services available that can be integrated directly into applications in the embedded network. We picked two examples which fit into the building automation scenario used in our prototypical implementation. Listing 3.5 shows an XML document from a weather service provided by the Norwegian Meteoreological Institute[1]. It contains the sunrise and sunset times on February 1st 2010. Such information can be used to automatically open or close the jalousies in an office building or a private home, even if no brightness sensors are available. The website of the Norwegian Metereological Institute contains interfaces which provide a fine grained access to specific weather information and which can be used directly by embedded devices. Most other weather services offer only a single interface for retrieving all weather information for a given location. This information contains forecasts for several days, various supplemental information and sometimes even

[1]http://yr.no

weather maps, resulting in a XML document that is simply to large for embedded networks. In this case, an integration is only possible by extracting a subset of the contained information, e.g., with an XPath expression.

```
<astrodata xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://api.yr.no/weatherapi/sunrise/
                                   1.0/schema">
  <meta licenseurl="http://api.yr.no/license.html"/>
  <time date="2010-02-01">
    <location latitude="48.264473"
              longitude="11.668839">
      <sun rise="2010-02-01T06:42:03Z"
           set="2010-02-01T16:12:17Z">
        <noon altitude="24.7122690084228"/>
      </sun>
      <moon phase="Waning gibbous"
            rise="2010-01-31T18:07:15Z"
            set="2010-02-01T07:33:52Z"/>
    </location>
  </time>
</astrodata>
```

Listing 3.5: Message From an External Weather Web Service

Listing 3.6 shows an example calendar entry using the xCal format, a XML representation of the widespread used iCal format. Such a document can be delivered by most calendar applications used nowadays. In a future building automation scenario, such calendar information can be used to dynamically prepare meeting rooms based on occupancy. In this vision, a meeting room is supplied with all calendar entries that take place in this room. Based on this information it can dynamically adjust the heating, cooling and lighting of the room or perform related optimizations.

```
<iCalendar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.ietf.org/internet-drafts/
           draft-ietf-calsch-many-xcal-01.txt"
    xsi:schemaLocation="http://www.ietf.org/internet-drafts/draft-
           ietf-calsch-many-xcal-01.txt xcal.xsd ">
  <vcalendar version="2.0">
    <vevent>
      <uid>1234</uid>
      <dtstamp>20100901T130000Z</dtstamp>
      <dtstart>20100903T163000Z</dtstart>
      <dtend>20100903T190000Z</dtend>
      <summary>JourFixe</summary>
      <location>Room 12.132</location>
    </vevent>
  </vcalendar>
</iCalendar>
```

Listing 3.6: Message From an External Calendar Web Service

It is hard to estimate what kind of Web services will be integrated into future embedded networks due to the vast number of interaction possibilities. We think that the two examples mentioned above are good representative w.r.t. the expected complexity of the documents and the amount of information contained in the documents used for communicating with these Web services.

3.2.2 Evaluation

We encoded the documents for the various test cases using different encoding techniques. Table 3.1 shows the resulting document sizes. For the gzip compression tests we used the built-in gzip support from Java SE 6. The WBXML tests were performed using the kXML library (version 2.2.2)[85], the Fast Infoset tests using the Java implementation provided by the GlassFish project (version 1.2.7)[45]. The EXI implementation was provided by the EXIficient project (version 0.4)[37]. The ASN.1 schema definitions were created manually based on the XML Schema definitions using the ASN.1 editor from Objective Systems (build 2010/01/29)[113].

To get comparable results, we removed the references to the XML Schema definitions for all encoders aside from EXIficient. These references are of no value if the encoding technique does not exploit XML Schema information and increase the message size considerably. EXIficient will not encode these references because the structure of the XML Schema documents is implicitly known from the grammar.

Table 3.1 shows that EXI-SA and ASN.1 provide massive savings compared to all other encoding techniques. Especially for small documents, such as the documents used in test cases 1 and 2, this difference is obvious. The documents created by EXI-SI and ASN.1 have a size that is only a tenth or less of the size of documents created with other encoding techniques. The reason for this big difference is that the documents used in the first test cases have a very bad markup/content ratio. The actual content of these messages are simple numeric values, which can be encoded in very few bytes. The XML markup is more than 100 bytes. The gap is closer if the markup/content ratio is better, e.g., when using the "Complex" test case or the "List" test case. The documents used in these tests encode fairly large data values (each EPC code alone is 12 byte long) and the encoding overhead for markup becomes less significant. Nevertheless EXI-SI and ANS.1 provide still a considerable reduction of message sizes.

To interpret the numbers shown in Table 3.1 correctly, one has to consider the boundary conditions imposed by embedded networks. In a real world environment, message sizes have to be kept very small. Bit error rates are comparably high using IEEE 802.15.4 networks. The larger a packet is the higher is the probability of a bit error, which requires a retransmission of the whole packet. TinyOS for example uses a default maximum payload size of 29 bytes. To achieve a good response time, it is important that messages exchanged between the services of a control application fit into a single network packet. These messages typically have a low complexity (one or more numeric values) but are often exchanged with high frequency. Test cases 1 and 2 represent these messages. As the numbers show, only EXI-SI and ASN.1 can provide the required compactness.

The tests cases for the interaction with the weather Web service and the calendar

	XML	GZip	WBXML	FI	FI-Dict	EXI	EXI-SI	ASN.1
Single								
short	130	129	81	67	31	46	2	1
medium	132	131	83	70	34	48	3	2
large	136	131	87	74	38	52	5	4
Multiple								
short	149	136	97	78	36	55	3	1
medium	152	138	100	82	40	58	5	4
large	159	139	107	90	48	65	9	8
Complex	263	195	201	174	117	139	46	34
List	1074	306	793	394	331	292	267	205
Weather	456	287	410	360	268	295	74	n.a.
Calendar	419	267	338	297	149	241	93	n.a.

Table 3.1: Message Size in Bytes for Different Encoding Techniques

application were not performed for ASN.1 because we could not find an (affordable) tool that provides an automated mapping between XML Schema definitions and ASN.1 definitions. According to the other numbers, the expected size of the ASN.1 documents will be very close to the size reported for EXI-SI. The resulting size for both documents is below 100 bytes what is only half of the size provided by other encoding techniques. Although these messages do not fit into a single network packet, their size is still small enough to be processable on resource constrained devices - assuming that such messages do not have to be processed very frequently.

3.2.3 Summary

Summing up, EXI-SI and ASN.1 are both suitable encoding techniques for embedded networks. The question is, which one is better suited for embedded networks? We chose to use EXI-SI because of two reasons: first EXI provides a cleaner mapping of XML documents. We assume that interactions with external Web services are common for future embedded networks, and XML support therefore is a key requirement. It is even likely that future Web service stacks will offer EXI support out of the box to increase the performance of Web service interactions (at least this was the motivation that led to the development of EXI). In this case, a conversion between EXI and plain XML is not required anymore. The second benefit of EXI is the possibility to encode schema deviations. We already gave a motivation how this feature can be used to provide an efficient encoding of commonly used values based on enumerations. The ϵSOA platform makes heavy use of this feature to create compact service descriptions and related documents that have to be stored and processed by nodes in the embedded network.

3.3 Hardware Description Language

Each node in the ϵSOA platform is described by a hardware description document. The hardware description is based on the hardware model already introduced in Section 2.3.1. It contains information about the resources available on a node, such as flash storage capacity or the amount of RAM available on a node. The hardware description document contains all static information, i.e., information that is independent from environmental influences.

3.3.1 eHDL Format

Variable information, such as link characteristics, is collected during runtime and dynamically added to the hardware model. The hardware description document uses a simple XML format we call embedded Hardware Description Language (eHDL). The corresponding XML Schema definition can be found in Appendix B.1, an example eHDL document is shown in Listing 3.7.

```
<ehdl:hardware xmlns:ehdl="http://in.tum.de/eSOA/hardware"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://in.tum.de/eSOA/hardware ehdl.xsd ">

  <!-- TMote Sky / TelosB Configuration -->
  <!-- 10k RAM -->
  <resource name="RAM" value="10240" />
  <!-- 40k ROM -->
  <resource name="ProgramMemory" value="40960" />
  <!-- 128k Flash -->
  <resource name="Flash" value="1048576"/>

  <!-- reference to attached device -->
  <hardware>100275</hardware>

  <!-- location information -->
  <property name="Floor" value="2" />
  <property name="Room" value="54" />
</ehdl:hardware>
```

Listing 3.7: Example eHDL Document for a Relay

The listing shows a hardware description for a node with 10k RAM, 40k program memory and 1MB of flash memory (the hardware parameters of a TMote Sky / TelosB node). Assume a light switch is attached to an I/O pin of the node. This information is specified in the hardware parameter. Additional management information concerning the location of the node is specified with two properties in the description. This information can be used by development tools to filter and search nodes.

As already outlined in Section 3.2, EXI can be used to efficiently encode enumerated types. Enumerations can also be used to efficiently encode commonly used values, e.g. the property names used in the hardware description. If a property name is used in the document that is not contained in the document, the schema deviation handling in EXI can be used to encode this value. Due to this mechanism it is possible to efficiently encode common values in EXI and at the same time preserve the flexibility to encode arbitrary values. Using a schema informed EXI encoding and enumerations, the example hardware description has a total of 23 bytes.

The eHDL document comprises three main parts: resources, hardware descriptions and properties. Resources describe the basic resources offered by a node which are consumed by the services running on this node. We currently use the following three resources: program memory, RAM and flash capacity. All resources describe the free amount of the resource after the installation of the ϵSOA middleware on the given node. This free capacity can be used by services running on a node. Because many microcontrollers use a Harvard-architecture, i.e., separated memory areas for code and data, we distinguish between the available amount of data memory (RAM) and the available amount of program memory. The flash can be used as long-term persistent storage. The flash also supports the storage of fairly large data sets compared to the very small amount of RAM available on many microcontrollers.

In most cases, a node will already be shipped with a set of hardware services that provide access to the sensor and actuator devices available on a node. If a node possesses some general hardware capabilities, such as a relay that can be used to turn on or off arbitrary devices, the hardware description section can be used to specify which devices are attached to the node. This information can be added by the Installer during the installation and configuration of the node. This information can be used to install corresponding hardware services on the node, when the node is connected to the network. The hardware device is specified with a hardware identifier (in our demonstrator we use a 64bit numeric identifier), which can be used to look up a corresponding hardware service from a repository.

The last part is the properties section. Properties allow storing arbitrary information as name, value pairs. A typical example is management information (e.g. numbers taken from an installation planning tool) or location information (e.g. GPS coordinates or room numbers). This information can be used to filter nodes or search nodes with specific properties and helps managing large scale installations. In Section 5 we will describe some possibilities how this information can be used by management and monitoring tools.

3.3.2 Domain Specific Description Models

We developed the eHDL description as a basic example format to showcase the functionality of hardware descriptors. In some domains, specific description standards have already been developed, e.g., the Electronic Device Description Language (EDDL)[2] in the automation domain. These formats can also be used to store the information contained in the hardware model. The only prerequisite is that the information can be represented compact enough to fit on an embedded device (if the format specifies an XML based encoding EXI can be applied, too).

3.4 Service Description Language

An important building block of service oriented architectures is the service description language. The service description provides the information required for the selection of and the interaction with services. In the first part of this section, we will take a look at the service description used for describing Web services: the Web Services Description Language (WSDL)[159]. WSDL has some drawbacks that prohibit its use in the context of embedded networks. We will present these drawbacks and an adapted service description language for embedded networks in part two of this section.

3.4.1 WSDL

In the Web service domain, the Web Service Description Language (WSDL) is used to describe services. The aim of the authors of WSDL was to create a language that can be used for "describing network services as collections of communication endpoints capable of exchanging messages"[159]. Due to this generic approach WSDLs can be used to describe services in a broad range of service oriented systems, including the ϵSOA platform.

A WSDL document comprises 6 parts. The *types* section allows defining data types. The *message* section provides an abstract definition of exchanged messages. Each message may comprise multiple parts, which are each mapped to one of the data types defined in the types section. *portTypes* can be used to specify communication end points with a specific message exchange pattern (one way or request/response interactions). The concrete protocol used to access a portType can be specified in the *binding* section. The address of a binding is specified in a *port*. *Services* finally can be used to bundle associated services.

WSDLs are not well suited to express the semantic information included in the ϵSOA service model. This weakness of WSDL is well known and there are some approaches to add this functionality to WSDL, such as Semantic Annotations For WSDL (SAWSDL)[156] or Web Service Semantics (WSDL-S)[157]. A problem is that none of these approaches is well established in practice and it is unclear which -if any- approach will be used in the future.

Because of the lacking support for semantic information and the bloated description format provided by WSDL, we did not use WSDL for describing services in the ϵSOA platform. Instead we designed a stripped down version of WSDL that is tailored to the needs of embedded networks and can be efficiently encoded using binary XML techniques. We call this language Embedded Service Description Language (eSDL). eSDL was designed in a way that allows an easy bi-directional mapping to WSDL. Developers already familiar with WSDL will find the same building blocks with the same semantics in eSDL, however the representation is different to allow a better compression with binary XML techniques. The only difference is that eSDL inherently supports the annotation of semantic information, which cannot be represented in WSDL (but one can easily define a mapping to one of the semantic annotations formats for WSDL), and that the expressitivity of eSDL is limited to a single binding per service. An eSDL can be converted to a WSDL for external Web services that are interested in interacting with services in the embedded network. A conversion from WSDL to eSDL is possible, too. In the latter case, the user has to add semantic information during the conversion process - if it is not contained in one of the already mentioned extensions of WSDL, which support semantic annotations.

3.4.2 eSDL

A lot of the complexity of WSDL descriptions can be attributed to the built-in flexibility. The WSDL specification provides a generic description of services, regardless of the used message protocol. A concrete service may use one or more bindings to specific protocols, such as the SOAP binding. This flexibility can be desirable in the Web environment to provide services that can interact using different communication protocols. The vast majority of Web services nowadays uses the SOAP binding. The flexibility provided by different bindings is therefore only very rarely used in the Web service domain. In the embedded domain, this flexibility is not needed at all. A lot of effort is currently done to design communication protocols that have a footprint that is small enough to fit on nodes with a few kilobytes of RAM. The overhead for supporting multiple different protocols is simply too high in such an environment. As a consequence, services in an embedded network communicate using one specific message protocol and the flexibility for specifying different bindings is not needed.

Another problem of WSDL that prohibits its use in embedded networks is the verbose description format. WSDL makes heavy use of string based references between the individual parts of the definition: services refer to ports, ports to messages, messages to datatypes, etc. These references consume a considerable amount of space, even if binary XML techniques are applied. The reason for this is that, even if dictionary based encodings are used, each reference has to be encoded at least once in the message. The dictionary will quickly require several ten's of bytes. Using very short strings for these refrences is typically not practicable. The readability of WSDL documents heavily depends on the usage of "sound" names for messages or ports. A WSDL that defines a port that uses message "1" as input, which has data of type "43", and produces a message of type "3", which has data of type "44", is hardly understandable by a human. In many cases the names used for these references also imply semantic information, such as the purpose of an operation, and cannot be replaced by numerical values.

An eSDL document uses the following concepts

- **Service** A service offers one or more Operations
- **Operation** Each operation defines a functionality offered by a service. Operations consume and/or produce messages.
- **Message** A message specifies the data being communicated based on a type system

eSDL uses nesting to avoid the references needed in WSDL. The Service element directly contains Operation elements as children. Like in WSDL, each Operation specifies a message exchange pattern based on the occurrence and order of Input and Output messages. The supported patterns are the same as in WSDL, i.e., One-way (input only), Request-response (input followed by output), Solicit-response (output followed by input) and Notification (output only). The messages directly contain the data type definitions. We will present eSDL in detail using an example description, which is shown in Listing 3.8. The XML Schema defintion for eSDL can be found in Appendix B.2.

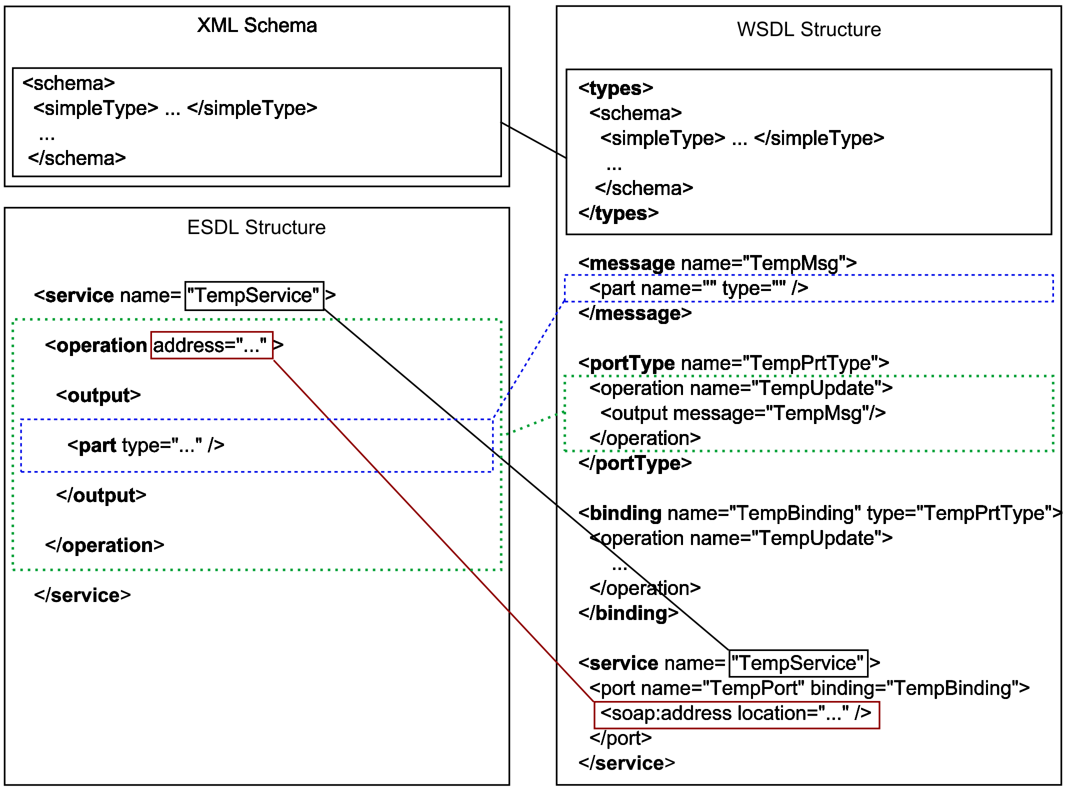
```
<esdl:service xmlns="http://in.tum.de/eSOA/service"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://in.tum.de/eSOA/service esdl.xsd "
  id="TemperatureService" >

  <operation address="1" mep="out-only">
    <output>
      <parameter measurandtype="Temperature" datatype="short">
      </parameter>
    </output>
  </operation>
  <operation address="2" mep="in-out">
    <input>
      <!-- message is used as trigger and contains no data -->
    </input>
    <output>
      <parameter measurandtype="Temperature" datatype="short">
      </parameter>
    </output>
  </operation>
</esdl:service>
```

Listing 3.8: Example eSDL Document for a Temperature Sensor

The eSDL description in Listing 3.8 describes a temperature sensor. The sensor offers an operation for the delivery of periodic measurements (operation “1”) and an operation for a request/response retrieval of a measurement value (operation “2”). Operation “1” uses an out-only message exchange pattern, i.e., delivers solely output data and accepts no inputs. Services interested in periodic measurements can subscribe to the temperature sensor and will receive measurement values from this output of the service. Operation “2” can be used to fetch temperature measurements on demand. Whenever a request is received, a measurement is performed and the measured temperature is returned in the response document. The “measurandtype”, “datatype” and common properties can be encoded efficiently using enumerations.

The *Types* section known from WSDL is moved to a separate XML schema document. This has a technical reason. The currently available binary XML en-/decoders are not capable of parsing an XML document based on schema information defined in the same document. Instead, all schema information has to be kept seperate from


XML Schema
<schema>
<simpleType> ... </simpleType>
...
</schema>
ESDL Structure
<service name="TempService">
<operation address="...">
<output>
<part type="..." />
</output>
</operation>
</service>
WSDL Structure
<types>
<schema>
<simpleType> ... </simpleType>
...
</schema>
</types>
<message name="TempMsg">
<part name="" type="" />
</message>
<portType name="TempPrtType">
<operation name="TempUpdate">
<output message="TempMsg"/>
</operation>
</portType>
<binding name="TempBinding" type="TempPrtType">
<operation name="TempUpdate">
...
</operation>
</binding>
<service name="TempService">
<port name="TempPort" binding="TempBinding">
<soap:address location="..." />
</port>
</service>

the encoded document. A service description therefore comprises two documents: an XML schema document containing the type definitions and the eSDL document itself. If the service uses only the built-in data types of XML schema, the eSDL alone is sufficient.

Service descriptions based on eSDL can be encoded very efficiently using EXI with schema informed encoding. The service descriptions for the services used in our demonstrators have a size smaller than 50 bytes (26 bytes for the document in the example).

3.4.3 Comparison of WSDL and eSDL

Figure 3.4.3 illustrates the correlation between WSDL and eSDL elements (to keep the example concise, the documents have been stripped down to the basic structure). Services in the ϵSOA platform support only one operation per PortType. The *PortType* element known from WSDL can therefore be omitted and the *Operation* elements can be nested directly into the *Service*. To avoid references, the *Message* definitions are directly nested into the Operations. As we already mentioned, all services in the ϵSOA platform support only one pre-defined binding. The *Binding* element is therefore unnecessary in eSDL. Like the binding, the addressing scheme is fixed in the ϵSOA platform. To further reduce the complexity, the addressing information provided by the *Port* element of the WSDL can be added directly to the Operation.

3.4.4 Summary

In this section we described eSDL, the service description language used in the ϵSOA platform. We chose to design a new description language over using WSDL due to two main reasons: to achieve a more compact descrption format and to offer built-in support for semantic annotations regarding the purpose of services and the type of measured data. eSDL was designed in a way that allows an easy transformation to and from WSDL to foster the integration of embedded and Web services and to provide an intuitive description language for developers already experienced in Web service development.

3.5 Embedded Service Choreography Language

The composition of services to applications has been studied for quite a while in the Web service domain. Two major composition paradigms have been developed for Web services: service orchestrations and service choreographies.

A *service orchestration* uses a central coordinator that orchestrates the interaction between different services. The most prominent orchestration language nowadays is the Web Service Business Process Execution Language (WS-BPEL)[112][2]. By using an orchestration language, the developer can specify execution workflows with basic programming constructs such as branches and loops. Furthermore, orchestration languages typically offer constructs for accessing external Web services, handling XML data, session and transaction management, etc. An orchestration language can be seen as a high-level programming language that uses Web service invocations instead of method calls for performing complex processing tasks.

A *service choreography* on the other hand has no central coordinator and is executed completely decentralized. Using a service choreography, a developer creates an application by defining the communication between individual services. He defines a set of peer-to-peer communication links between services, which are used to propagate data from one service to another. The most prominent Web service choreography languages are the Web Service Choreography Interface (WSCI)[160] and the Web Service Choreography Description Language (WS-CDL)[164].

The stream based processing model used in the ϵSOA platform essentially is a choreography based service composition approach. The motivation for using a choreography rather than an orchestration was twofold. First, choreographies avoid the creation of single points of failure. If the node executing an orchestration fails, the whole application ceases to work - even if redundant hard- and software is available. The second reason is that choreographies are easier to scale than orchestrations. An orchestration introduces a single point of control which is involved in all interactions between services. This is problematic for distributed control applications because the node executing the orchestration will have to process large amounts of sensor data. By using a choreography, this processing load can be distributed over multiple nodes. Furthermore, service choreographies are much better suited to exploit locality. With a choreography, it is possible to place data consuming services close to data producing services in order to reduce the overall communication overhead. In an orchestration this is not always possible. Assume we have two sensor service producing high volume data streams. These streams are filtered by two logic services and afterwards submitted to a database service that stores the measurements for later processing. Using a choreography, both filters can be placed close to the sensor devices and only the filtered streams have to be transmitted to the database service. Using an orchestration, the sensor data first has to be transmitted to the orchestration workflow and is then forwarded to the filter. Because the orchestra-

[2]Other examples of business modeling languages are the now deprecated Business Process Management Language (BPML), the predecessors of BPEL: Web Service Flow Language (WSFL) and XLANG, and the XML Process Definition Language (XPDL).

tion workflow is a single point of control, it can be placed either close to one sensor or the other, never both at the same time. This results in a much higher network utilization compared to the choreography based solution.

We will not present WS-BPEL in detail here and instead focus on the choreography languages WS-CDL and WSCI. A general overview of service composition standards can be found in Peltz[117], Bucchiarone et al.[9] and Srivastava et al.[144]. We will present a short summary of WS-CDL and WSCI here, a more detailed comparison can be found in Cambronero at al.[12].

3.5.1 Choreography Languages in the Web Service Domain

WSCI specifies an extension to WSDL that allows describing complex operations beyond the basic interface definitions possible with WSDL. WSCI specifies the interaction between Web services in terms of choreographed activities. Activities can either be simple WSDL operations or more complex structured activities that allow the execution of multiple actions in sequence, in parallel or in a loop. WSCI offers support for transactions by allowing the definition of correlation elements that are used to identify messages belonging to the same transaction and by providing exception handling and compensation mechanisms.

WS-CDL can be seen as an successor of WSCI. In WS-CDL interactions occur between participants. Each participant can be assigned one or more roles. Each role is identified by a name and may be assigned to an interface defined in a WSDL. If an interaction requires commitments from both endpoints, e.g., a purchase request that must be accepted by the vendor, the relationship construct from WS-CDL can be used to express this requirement. The communication between participants is handled via channels. A channel specifies how and when a communication between participants occurs. WS-CDL offers variables and tokens to store intermediate information. A token is a reference, specified in XPath, to a value that may occur in multiple messages but at different locations, e.g., a customer id. The heart of the WS-CDL specification is the choreography element. A choreography defines workunits which encapsulate activities. Workunits can be used to force an alignment, i.e., an agreement between multiple participants regarding the value of a variable. WS-CDL offers activities for parallel, sequential and conditional execution of activities, and an interaction activity. An interaction references a channel, a Web service operation, a “from” and “to” reference to a role, and some additional variables. Using these constructs, WS-CDL allows describing a Web service orchestration based on a set of service-to-service communications.

While WS-BPEL has emerged as a dominant and widely used standard for service orchestration, service choreography standards are not used on a widespread basis. One reason might be some inherent limitations of the languages as described by Decker et al.[26]. A limitation of WS-CDL and WSCI is that both languages are “not an executable business process description language”, as it is clearly stated in the WS-CDL specification. The intended purpose of both specifications is to allow the modeling of choreographies, the concrete realization is out of the focus of the specifications. There is some work that aimes at specifying an execution model for WS-CDL[42], however this work has just been started and it is unclear whether this model can be implemented efficiently on resource constrained devices. Another problematic aspect of WS-CDL and WSCI is the complexity of the generated XML documents. Even very simple choreographies will create huge documents due to a very bloated XML syntax and the definition of many business related concepts, such as the participants, roles and relationships in WS-CDL. These concepts are not required for specifying interactions in an embedded network. Another problem is the limitation of the communication pattern to peer-to-peer communications. The submission of single measurements to multiple consumers cannot be modeled in WS-CDL and multicast capabilities present in an embedded network cannot be exploited due to this limitation.

Due to these limitations, we decided to develop a new choreography language that is tailored to the needs of embedded networks, provides an efficient representation of the application model introduced in Section 2.3.3 and can be executed on resource constraint devices. We call this language embedded Service Choreography Language (eSCL).

3.5.2 Embedded Service Choreography Language

eSCL was designed to meet the following four design goals:

Compactness eSCL should be very compact to allow storing and exchanging choreography descriptions in resource constrained embedded networks.

Instance Based Choreographies The second goal was to exploit the service lifecycle model used in the ϵSOA platform. eSCL defines an choreography based on service **instances**. Service instances in the ϵSOA platform are explicitly instantiated and can be addressed using an instance identifier. This removes the need for correlation information as used in BPEL, WS-CDL and WSCI. If a stateful service should be integrated in a choreography, a distinct instance will be created that can be addresses directly from the eSCL choreography.

Modelling of Redundant Components The third design goal was to support the modeling of redundancy. Failure recovery in the Web service domain typically involves the execution of compensatory actions to achieve a consistent state between all Web services involved in a transaction. All standards targeting Web service compositions assume that the availability of the involved services is guaranteed by the Web service providers or that a replacement for a failed service can be found by issuing a lookup in a service repository - otherwise an error occurs and the composition fails. These assumptions do not hold for embedded networks. To provide a fast and resource efficient reaction to service failures, the ϵSOA platform allows the specification of redundant service composition that will be fully automatically activated whenever a composition fails. The preconfiguration of redundant components ensures that the replacement of a failed service instance will result in a meaningful service composition, i.e., a service composition that provides the desired outcome. Furthermore, the reconfiguration is simplified in a way that ensure it can be performed in a timely manner even on resource constraint devices. A design goal of eSCL was to support the modeling of such redundant compositions directly in the choreography language.

Graceful Degradation In some cases, a failure recovery is not possible, e.g., because there are no redundant devices available. If this service was critical for an application, the execution of the whole application should be stopped. If the remaining composition is still meaningful, the execution should continue. In the latter case, some adjustments in the behavior of the remaining service instances might be required to compensate the missing component. Assume the brightness sensor fails in a lighting application. In this case, the lighting application should not be deactivated. Instead, the control logic could deactivate the dimming of lights based on the measured brightness and always turn on the lights. This ensures the end user will still be able to turn on the lights, but will experience reduced functionality (graceful degradation). The implementation of degradation mechanims is domain and service specific. The ϵSOA platform offers a monitoring and notification interface that reports missing service instances to the remaining instances of a service composition. Based on this information, the instances can adjust their behavior. The fourth design goal for eSCL was to integrate concepts that allow the developer to specify under which conditions a graceful degradation or a deactivation of service compositions should be triggered .

3.5.2.1 eSCL Format

We will present eSCL by using the example description shown in Listing 3.9, the corresponding XML Schema definition can be found in Appendix B.3.

```
<escl:choreography xmlns:escl="http://in.tum.de/eSOA/choreography">
  <instances>
    <!-- switch -->
    <instance instanceId="3000" service="LightSwitch" />
    <!-- light logic -->
    <instance instanceId="3010" service="BasicLight" />
    <!-- redundant light logic -->
    <instance instanceId="3011" service="BasicLight" />
    <!-- bulb 1 -->
    <instance instanceId="3020" service="LightBulb" />
    <!-- bulb 2 -->
    <instance instanceId="3021" service="LightBulb" />
  </instances>

  <operationGroups>
    <!-- output of switch -->
    <operationGroup id="1">
      <operation instanceId="3000" operation="0"/>
    </operationGroup>
    <!-- redundant input for logic -->
    <operationGroup id="2">
      <operation instanceId="3010" operation="0"/>
      <operation instanceId="3011" operation="0"/>
    </operationGroup>
    <!-- redundant output for logic -->
    <operationGroup id="3">
      <operation instanceId="3010" operation="1"/>
      <operation instanceId="3011" operation="1"/>
    </operationGroup>
    <!-- input of light 1 -->
    <operationGroup id="4">
      <operation instanceId="3020" operation="0"/>
    </operationGroup>
    <!-- input of light 2 -->
    <operationGroup id="5">
      <operation instanceId="3021" operation="0"/>
    </operationGroup>
  </operationGroups>

  <dataStreams>
    <!-- switch to logic -->
    <dataStream streamId="1" source="1" drain="2"/>
    <!-- logic to bulb 1 -->
    <dataStream streamId="2" source="3" drain="4"/>
    <!-- logic to bulb 2 -->
    <dataStream streamId="3" source="3" drain="5"/>
  </dataStreams>

  <streamGroups>
    <!-- both bulbs are in one stream group -->
    <streamGroup>
      <dataStream streamId="2"/>
      <dataStream streamId="3"/>
    </streamGroup>
  </streamGroups>
</escl:choreography>
```

Listing 3.9: eSCL Example

The first part of an eSCL choreography is a definition of the used service instances. Each instance is uniquely identified by a numeric id. Additionally, each instance contains a reference to a service identifier that can be used to look up the interface definition of the instance. One or more of the operations offered by each of these service instances can be bundled into an *operationGroup*. An operationGroup may contain one or more *operations*. Each of these operations is identified by the instance id and a reference to the operation definition in the eSDL service description of the corresponding service. The operationGroups are used to define *dataStreams*. A stream references an operationGroup as source and an operationGroup as drain. If the operationGroup contains only a single operation, the ϵSOA platform will automatically install corresponding data stream between the in- and outputs defined by the corresponding operations. To model redundantly available instances, more operations can be added to an operationGroup. The ϵSOA platform currently supports only a comparatively simple redundancy model for operationGroups: operations are used in document order, i.e., the first operation mentioned in a operationGroup is preferred. If the corresponding instance is not available, the second operation is selected, and so on[3]. A possible extension is to add priorities to the operation references in an operationGroup.

If no operation in an operationGroup is available, the corresponding data stream is marked as malfunctioning. This information can be used in the last part of the choreography specification, the *streamGroup* definition, to specify when an application should be stopped. A stream group may contain one or more references to data streams. If all data streams contained in a stream group fail, the whole streamGroup is marked as malfunctioning and the application is stopped. This mechanism can be refined by using the optional count attributes of the streamGroup which allow to specify a minimum number of instances that has to be available. Data streams may be added to multiple streamGroups, which allows the definition of fairly complex dependencies between streams. The expressiveness offered by the streamGroup construct was sufficient to model all failure situations we encountered in our demonstrators so far. A possible extension would be to allow the specification of more complex rules, e.g., using event, condition, action (ECA) rules to support more fine grained control over the actions taken whenever an application fails. However, this ruleset must be designed carefully to allow an execution on embedded devices.

In the example, there are five instances, three instances of hardware services (a light switch and two light bulbs) and two instances of a light control software service. The example defines a redundancy relationship between both instances of the logic service. This is done in operationGroups 2 and 3, which each define instance 3010 as preferred instance and 3011 als redundant instance. If instance 3010 is not available (e.g. because of a node failure) the ϵSOA platform will automatically search for replacements in each operationGroup that contains instance 3010. It will then reconfigure the stream routers on the individual nodes and remove all entries belonging to instance 3010 and add new entries for instance 3011. A more detailed description of the failure compensation algorithm can be found in Section 4.1. The light bulbs have no redundant replacements. If one of the light bulbs fails, e.g. instance 3020, the corresponding streamGroup becomes unavailable and the corresponding data stream is marked as failed. The example specifies a stream group containing data streams for both bulbs. As long as one of the bulbs and therefore also the corresponding data stream is available, the application will not be stopped. If both bulbs fail, or the data stream between the switch and the logic fails, the application will be stopped.

Using a schema informed EXI encoding, the example document has a size of 104 bytes, which is small enough to be stored and processed even on resource constraint

[3]The selection process is a bit more complex because a service may offer multiple operations simultaneously. In this case, a replacement is selected that satisfies all affected operationGroups

microcontrollers.

3.5.3 Summary

In this section we introduced eSCL, the XML based description language for application compositions used in the ϵSOA platform. Its design was motivated by the observation that existing choreography languages are not applicable for embedded networks. eSCL provides a compact representation of service composition and supports the modelling of redundant components out of the box. This information is used by the failure recovery mechanisms presented in Section 4.1 to provide an autonomous compensation of node failures.

3.6 Communication Module

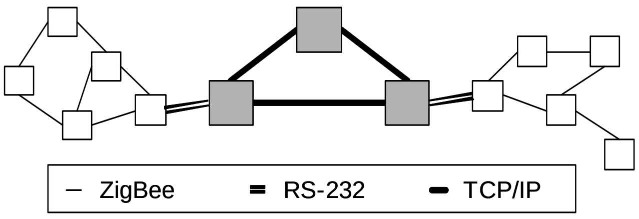
The communication layer is a crucial part of embedded networks. Communication costs constitute a major part of the overall power consumption, especially if wireless communication media are used. A basic requirement for efficient data processing in embedded networks therefore is a suitable communication infrastructure. In this section we will motivate and present the design of the network stack used in the ϵSOA platform, which is based on a modular design. The message payload transported with this stack are EXI documents, the used format and corresponding parser and encoder implementations are presented in Section 3.7.

3.6.1 Communication Requirements

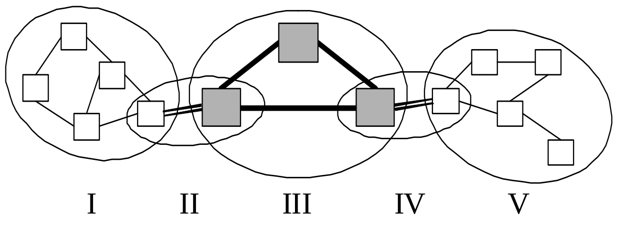
Embedded networks cover a broad range of application fields with different requirements concerning the communication between the individual nodes in the network. Because the characteristics of the underlying transport media and the envisioned application fields are diverse, it is unlikely that a single communication protocol can be designed which is suitable and efficient enough for all these scenarios. As a consequence a lot of specialized protocols are emerging which are tailored for a given application scenario or use case. Examples for such protocols for wireless sensor networks are: 6LoWPAN[63], which provides IP communication for low power wireless networks, Ad hoc On-Demand Distance Vector Routing (AODV)[148], which provides routing for ad hoc wireless networks, data-centric[51][65] routing protocols, energy aware routing schemes[135], and many more.

Given the variety of network protocols, the developer of an embedded network is faced with several challenges: first he has to choose a suitable protocol for his given application field. However, this choice should not be final. If the intended application scenario changes or a better suited protocol becomes available, the protocol should be exchangeable. This should be possible even if the new protocol uses a different addressing scheme and should not require a reconfiguration of all installed applications.

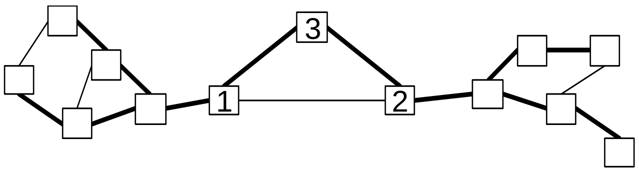
The second challenge are heterogeneous networks comprising several different protocols. A typical example for this scenario is depicted in Figure 3.2(a). It comprises a backbone of nodes connected via an Ethernet running the IP protocol and multiple sensor nodes connected via ZigBee to the backbone. In this scenario, three network types with different protocols are used, as shown in Figure 3.2(b): ZigBee for the communication between the sensor nodes (Subnets I and V), a RS-232 interface for the communication between the sensor nodes and the backbone nodes (Subnets II and IV), and TCP/IP for the communication between the backbone nodes (Subnet III). We will use the term “subnet” to refer to such clusters of nodes that communicate with the same network protocol. Assume the developer wants to run a simple monitoring application that calculates average values over the measurements of all sensor nodes. A possible solution is illustrated in Figure 3.2(c): Node 1 is used to calculate the average for the left ZigBee subnet, Node 2 for the right ZigBee sub-



(a) Scenario



(b) Subnets



(c) Data Flows

Figure 3.2: Heterogeneous Embedded Network

net. Node 3 is used to calculate the average of the intermediate results produced by Node 1 and Node 2. The corresponding data flows are shown as thick lines in Figure 3.2(c). During application development the developer should not have to worry about the message conversions required to facilitate a communication between the individual nodes, but should be able to work on an abstract view of the network that hides the underlying protocols and communication media. Nevertheless, a majority of the communication will occur between nodes in the same subnet and should not be hindered by a too complex abstraction layer with high overhead. Finally, the special characteristics of the underlying networks, such as bandwidth restrictions, have to be represented in the abstract network view in order to support an optimization of the placement of services and to avoid overload situations.

A third challenge is the heterogeneity of the involved networks. Some of the used network protocols may already offer reliable transmissions, either through a reliable communication medium or a reliable transport protocol, whereas other networks only offer unreliable transport mechanisms. We will refer to these capabilities as "features" in the remaining part of this work. A possible solution is to use an overlay network that offers all the required features. The major drawback of this approach is the inefficient resource usage. If some features are already implemented by an underlying protocol the overhead of a re-implementation should be avoided. Additionally, not all nodes may require the same features. Consider a network comprising temperature sensors which are used to provide long-term monitoring and do not require a reliable data transport, because the loss of some measurement values is acceptable. At the same time the network contains a fire detector which has to report fire alarms reliably. Given this scenario the network stack on the temperature sensing nodes does not have to provide a reliable transport protocol, but the stack on the alarm and the fire detector has to. To provide an efficient communication in this scenario, the network stack used on individual nodes has to be adaptable. It should provide the features required by the services running on the node, but not offer additional features if the provisioning of these incurs an overhead. Additionally, features already available in the underlying network should be exploited and not re-implemented by the network stack.

Messages exchanged via the communication layer can be grouped into two classes: plannable interactions based on data streams flowing between two services and ad-hoc interactions. Ad-hoc interactions typically occur during (re-)configurations or other administrative tasks. They can occur at any time and there is no information available regarding the characteristics of these interactions, such as data rates. There are some use cases for control applications that also perform an ad-hoc selection of their communication partners, especially for mobile devices. Nevertheless, the vast majority of applications executed in the embedded network will communicate via long-lived stream based interactions. An important property of data streams is that the source and sink of these streams is known a priori. Often the data rates are known, too. It is possible to transfer data streams using the same messaging protocol used for transmitting ad-hoc requests. But this is not advisable from an optimization point of view. A communication layer optimized for the transmission of data streams provides a much higher efficiency and reduced resource requirements compared to the solution based on a generic messaging protocol. Because data stream transmissions constitute a major part of message exchanges in the embedded network, the communication layer should provide an optimized handling for these data streams.

Summing up, the communication layer for embedded networks should provide the following features:

- Easy addition and removal of network protocols
- Support for communication over heterogeneous subnets
- Efficient communication within a subnet
- An adaptable network stack that can be tailored to application needs and exploits features provided by underlying protocols
- Support for data stream oriented communication
- Support for ad-hoc interactions

3.6.2 The ϵSOA Adaptive Network Stack

The ϵSOA network stack is based on two principles: modularity and re-use. A major design goal was to create a communication layer that introduces as little overhead as possible compared to existing network protocols. At the same time, the functionality of this stack should be scalable, i.e., small nodes with little processing capabilities should not be burdened with functionality not needed by the services running on the node. This can be achieved by a modular network stack that allows tailoring its functionality by adding or removing features depending on the needs of the services running on a node. To provide a resource efficient implementation of this stack, all features provided by the underlying network protocol should be re-used and not re-implemented in upper layers of the network stack.

The modular network stack shown in Figure 3.3 is based on a layered architecture. It comprises an *Abstract Network* layer which is located on top of the protocols used in the subnets. This layer provides a unified addressing scheme across all networks and provides modules for features not present in the underlying protocols. Based on the Abstract Network, the Data Exchange layer offers two communication interfaces: the *Stream Routing*, which is optimized for the transmission of continuous data streams, and the *Packet Routing*, which is optimized for message exchanges occurring during ad-hoc interactions. Additionally a *Cross-Layer Component* provides access to information gathered by the different layers, allowing the *Optimizer* to adapt the configuration of the network stack to the needs of a given application. We will describe the individual components of the network stack in detail in the following sections starting from the bottom with the Network Protocol Layer in the following section.

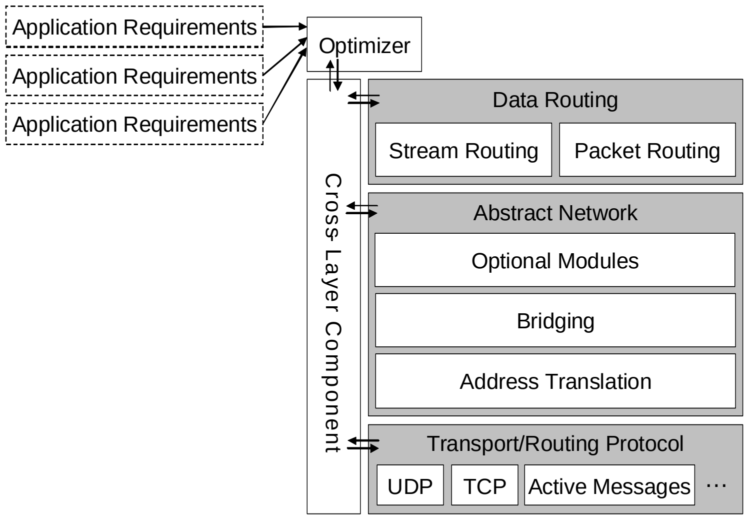


Figure 3.3: Architecture of an Adaptive Network Stack

3.6.3 Transport/Routing Protocol Layer

The bottom layer of the stack provides access to the transport or routing protocols used in the different networks, such as the UDP and TCP protocols used in IP based networks, the Active Messages used in TinyOS, the RS-232 interface for serial data transmission, etc. The minimum requirement for protocols that should be incorporated in the network stack is support for a unicast end-to-end communication between nodes. Potential candidates are therefore all protocols from the OSI-layers 3 and 4. Besides this basic functionality, many protocols offer additional features, such as multicast support, reliable transport, encryption, QoS guarantees, etc. These features are stored for every protocol. During the installation of an application, the Optimizer determines which of the features required by an application can be provided by the underlying protocols, and which features have to be provided through the installation of an additional module. The Cross-Layer Component allows the transport/routing protocols to publish topology information and link characteristics and to access application level information such as the data rates of streams, etc. The former information will be used to optimize the placement of services in the network, what is explained in detail in Section 2.6. The latter can be used by the transport/routing protocol to optimize the routing of packets in the network.

3.6.4 Abstract Network Layer

The functionality of the Abstract Network Layer resembles the functionality of network stacks known from overlay networks, such as Peer-to-Peer networks. However there is an important difference: The Abstract Network only handles the message routing across network boundaries. Messages sent within a subnet will be transmitted directly via the underlying transport protocol. Therefore the Abstract Network is not a full-fledged overlay network but can bee seen as a thin wrapper that allows communication across heterogeneous subnets. The rational for this decision are performance considerations. If communication occurs within a subnet, the Abstract Network incurs no additional overhead because all messages are sent directly via the underlying network protocol. If a packet is addressed at a node in another subnet, the packet is sent to a bridge node which converts the packet and injects the new packet in the other network. This is done by the *Bridging* component.

The *Address Translation* provides a unified addressing scheme across all subnets. It uses n-bit logical addresses for the identification of nodes, which comprise a network identifer (the first m bits) and a node identifier (the remaining $n - m$ bits). The number of bits used for addresses, n, and the distribution of these bits between the network and node part can be chosen during the code generation and deployment of the sensor network. This allows to reduce the network header size for small installations and support scenarios with a multitude of different subnets. However, only nodes stemming from networks with identical parameters for n and m can communicate seamlessly across network boundaries. If different parameters are used in two networks, a gateway that performs a network address mapping has to be used.

Address Assignment The Abstract Network currently supports two assignment schemes for logical addresses: static addresses and dynamic configuration. Static addresses are assigned during code generation and configuration of the network and remain unchanged at runtime. This allows creating very compact and small images for devices, which are not added to networks dynamically, e.g., switches, static sensors or similar devices. The second address assignment scheme is based on a DHCP-like dynamic configuration mechanism. New nodes request an address by sending a broadcast message after installation. A coordinator node in every subnet handles these requests and creates unique addresses for new nodes.

Address Resolution In order to send a packet to a remote node, the logical address of this node has to be mapped to the addressing format used by the underlying network protocol. This is done by the Address Resolution component. Similar to the address assignment, the address resolution can be performed either in a setup phase before the execution of applications is started, or dynamically at runtime. In the first case, the address resolution is performed when a new data stream is installed at the node and the network address of the destination node is stored in the Stream Router along with the routing information for a specific data stream. The dynamic address resolution uses a broadcast mechanism. Nodes can send a discovery broadcast for a logical address and will receive a reply from the corresponding node, which contains the network address.

Modules Optional modules can be plugged into the Abstract Network layer to provide additional features not present in the underlying transport or routing protocols. Consider an application requiring a reliable message transmission between two services. If this application is running on top of a TCP/IP network, the TCP protocol already provides this feature out of the box. If the underlying protocol supports only unreliable transmissions, e.g., UDP/IP, an additional reliability module is installed in the network stack at the sender and at the receiver. This module can extend the message header with additional fields in order to provide the desired functionality.

Modules are organized as a stack, i.e., on the sender side the message is passed to all modules in a top-down, on the receiver side a in bottom-up manner. Modules higher up in the stack can therefore treat modules lower down in the stack as black boxes. Assume an application requires a connection between two services, which offers reliable transport and data encryption. In this case the reliability module is installed below the encryption module in the network stack. At the sender, the encryption module is called first, which can encrypt the message payload. Subsequently the reliability module is called, which extends the message header with data needed to perform the reliable transport, e.g., a sequence number. The modules on the receiver side will be invoked in reverse order, i.e., first the reliability module and second the encryption module. If a transmission problem occurs, e.g. through a lost packet or a duplicate packet, this situation will be handled transparently by the reliability module and will not be visible to the encryption module.

We implemented some basic modules to demonstrate the feasibility of the modularization. Some of the modules are very simple and should be seen as a proof of concept implementation, modules used in a productive environment will most likely be more complex. The supported modules are:

Reliability Module The Reliability Module supports per-packet reliable transport. Outgoing packets are assigned with a unique sequence number and stored at the sender. The receiver submits an acknowledgement for every received packet. The next packet of the stream is sent after the acknowledgement of the previous packet is received. If the acknowledgement is not received in a configurable period of time, the packet is treated as lost and resubmitted by the sender. This mechanism can be implemented very efficiently on resource constrained devices, because only one packet has to be stored per data stream. It is well suited for a reliable transmission of control signals in the embedded network. The drawback is a poor performance compared to stream based protocols, such as TCP, if large data volumes have to be transmitted.

Fragmentation Module The Fragmentation Module is required if the underlying network protocol is packet based and the transmitted payload exceeds the packets capacity. The Fragmentation Module will break the message into peaces that fit into the network packets and reassemble the original payload at the receiver.

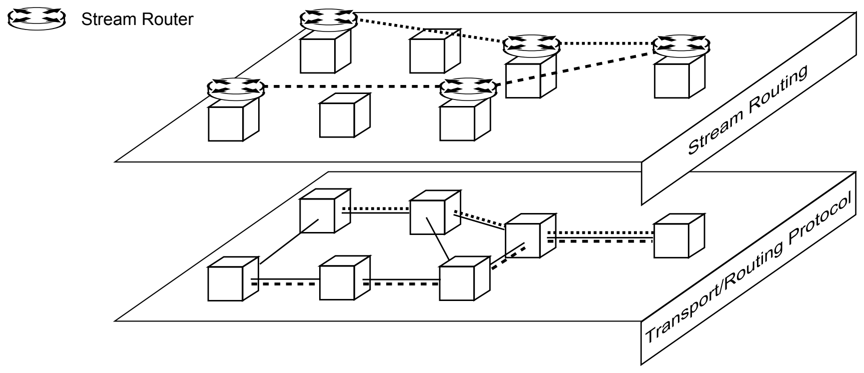


Figure 3.4: Layered Routing

Encryption Module Providing secure communication and authentication mechanisms for embedded networks is an open research area. The resource constraints on the devices often prohibit the use of public key infrastructures and asymmetric encryption algorithms known from other distributed systems. Often it is impossible to store long cryptographic keys on the nodes or perform complex calculation required by algorithms such as RSA. The Encryption Module should be seen as a demonstration how encryption support can be added to the network stack, once a suitable mechanisms has been developed. It supports a simple stream based symmetric encryption algorithm and is based on the assumption that a symmetric key pair is installed on each node and a trusted coordinator in the network. The coordinator authenticates new nodes and generates session keys for the communication between nodes in the network. Falk and Hof[38] describe a security design that can be used to create and manage such session keys.

3.6.5 Stream Routing Layer

The ϵSOA platform supports the transmission and management of data streams through the Stream Routing component of the network stack. The Stream Routing component operates on top of the network protocols (and the abstract network layer) used in the embedded network. The basic idea of this approach is depicted in Figure 3.4. The Stream Routing Layer uses data Stream Routers that are installed on each node in the embedded network. A data Stream Router directs incoming data stream elements to one or more targets. These targets can either be service instances on the local node, or data Stream Routers at a remote node. The target of an incoming data stream element is determined based on the id of the data stream it belongs to. To perform this task, the Stream Router contains an internal routing table that stores a list of targets for each data stream id.

The Stream Routing Layer handles the high level distribution of data streams and messages in the network. It specifies the sink nodes of data streams and provides functionality to split data streams at nodes. Note that the source and sink of data streams do not have to be neighboring nodes, but can be any node in the embedded network, even nodes from different subnets. A data stream only specifies on a high level that the data has to be transmitted from one node to another. The transmission of individual packets of the data stream and the selection of a suitable route in the underlying network is handled by the Network Protocol Layer. This layer relies on one of the various network protocols available for embedded networks to perform the actual transport of data between two nodes.

The layered approach is beneficial from a modeling point of view: the Stream Routing Layer provides a high level overview of the data transmissions in the network. It shows the data flows in a way that is understandable for an application domain expert and safely hides details regarding the underlying network protocols. The expert does not have to deal with the clustering of nodes or different network protocols used in different subnets. The expert can even influence the data transmission based on this high level of abstraction, the required reconfigurations are performed transparently by the ϵSOA platform.

Besides this modeling aspect, the Stream Routing Layer also possesses some interesting properties from an implementation and optimization point of view, which are discussed in more detail in the following sections.

3.6.5.1 High Level Routing of Data Streams

The Stream Routers control the flow of data streams through the system. They can be used to direct streams over specific nodes or to avoid nodes which are overloaded or have low energy resources. This can be done by adding intermediate Stream Routers (anchor points) to a data stream. The anchor point is a data Stream Router that simply takes the incoming stream and dispatches it to the final destination or the next anchor point in line.

The layered routing used in the ϵSOA platform was motivated by the observation that the routing of data streams and the routing of network packets operate on different levels of abstraction. The Stream Routing Layer is configured based on information from the application layer. During its configuration, data rate estimations, information about the long term behavior of applications and data streams, and a global overview over the topology and characteristics of the embedded network are available. The routing decisions in this layer are often based on long term optimization goals. Assume that in the example mentioned above, the node in the bottom right corner of the picture is battery powered whereas all other nodes have an unlimited power supply. In this case, the alternative route using the upper left node as anchor can be beneficial to maximize the lifetime of node x.

The routing decisions for individual packets are performed by the protocols in the Network Protocol Layer. These protocols have timely access to network topology changes, such as broken or congested links, failed nodes or new nodes added to the network. They can react quickly to these changes and adapt the routing of packets in a way that ensures the packet arrives at its destination. Compared to the Data Routing Layer, the routing decisions taken at the Network Protocol Layer are focused on a much smaller time scale. Furthermore, nowadays network protocols typically do not exploit information from the application layer, even if this information is made available through a cross layer interface. Therefore they cannot perform global optimizations regarding the routing of multiple data streams or optimizations based on the long term behavior of applications.

Note that many of the global optimizations can also be performed fully automatically by the system. The placement algorithm described in Section 2.6 can be extended to include such optimizations. The gains achievable through these optimizations depend on the application scenario.

There are some points that have to be taken into account when using the Stream Routing Layer to influence the data dissemination in the embedded network. Because every node executes a Stream Router, every node can be used as a anchor point for data streams. The Stream Routing Layer can therefore be used to control the flow of data streams on a very fine grained level. However, it is rarely beneficial to specify every single hop of a data stream via anchor points. In this case, the routing protocol used in the underlying Network Protocol Layer degenerates to a single hop transmission protocol. All functionality provided by the routing protocol is lost, such as the quick response to network changes. In a typical situation it is sufficient to specify a single anchor point for a data stream in order to redirect the stream.

A second point is that there is no guarantee that a network packet will not pass a specific node or use a certain link. The ultimate routing decision is still performed by the network protocol. Theoretically, it may choose an arbitrary path in the network to deliver the message. In a practical setting, this is rarely a problem, because the currently available network protocols behave quite predictable and will not arbitrarily change their routing decisions. Nonetheless, the Stream Routing Layer should not be used to perform security based optimizations. It cannot be guaranteed that a data stream will only use specific (secure) links. Such routing guarantees have to be provided by the network protocol, or cryptographic measures have to be taken to prevent access to the data stream.

Currently there is a lot of work aiming at the optimization of network protocols. We are looking at ways to push the information available at the application layer down to the network protocols. Based on this information, a smart routing protocol could perform some optimization tasks on its own. The routing functionality offered by the Stream Routing Protocol should be seen as an opportunity to achieve this functionality with the network protocols currently available. If future network protocols offer built-in support for these features, the ϵSOA platform can exploit this functionality by simple removing all anchor points and giving the network protocol full control over the transmission of data streams[4].

[4]It is not clear that such a protocol can be built. Especially the global optimizations are very hard to perform given the resource constraints on the embedded devices and would most likely require a distributed solution using the resource capacities of multiple nodes. As a consequence, the high level optimizations provided by the Data Stream Routing layer might still be beneficial in future embedded networks, despite the advances in the functionality provided by the network

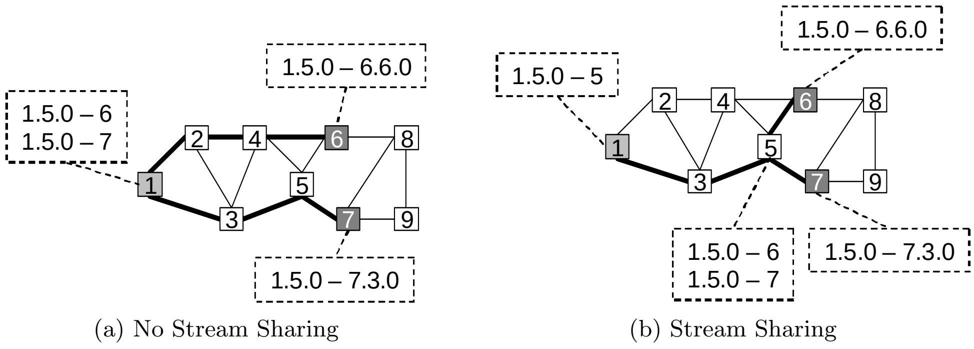


Figure 3.5: Data Stream Rounting

3.6.5.2 Data Stream Sharing

Whenever data is consumed by more than one service, data stream sharing can provide a reduction in the required network traffic. Instead of creating a distinct data stream for each source, it is often beneficial to create a single data stream that is split at an intermediate node. Such an example scenario is depicted in Figure 3.5(a). The data produced by node 1 is used by two other nodes, 6 and 7. The thick lines indicate the two resulting data streams routed through the network. A possible optimization which reduces the network utilization is shown in Figure 3.5(b). Instead of sending two distinct data streams to nodes 6 and 7, a single stream is sent to node 5. There it is split into two streams targeted at nodes 6 and 7. The dashed boxes in Figure 3.5(a) and 3.5(b) show the routing tables for the example. Every row contains one routing entry, the part on the left side of the "-" is the stream identifier, i.e., the source address, the part on the right side is the target address. Like the source address, the target address consists of a node address, an instance id and a port number. Because the instance id and port number are only needed at the target node, the Stream Router stores only the node address for remote services. The routing table of node 1 in Figure 3.5(a) can be read as: transmit the data produced by service instance 5 port number 0 to nodes 6 and 7. At node 6 the routing table specifies that the data should be processed by service instance 6 port number 0. The shared use of the data stream produced by node 1 can be achieved with the routing tables shown in Figure 3.5(b). Instead of two different streams, node 1 only creates a single stream that is targeted at node 5. Because the stream identifier is contained twice in the routing table, the stream is split at node 5 and sent to both nodes, 6 and 7. The routing tables at nodes 6 and 7 can remain unchanged.

Note that the splitting does not necessarily have to be performed at a dedicated node in the network, but can also be done at one of the sink nodes. In this case, the elements are dispatched to the service instance interested in the data and another

protocols used in these networks.

sink node. This can be done for an arbitrary number of sinks, resulting in a single data stream flowing through all sink nodes. The data Stream Routers mentioned above offer built-in support for these scenarios. Every node in the network runs a data Stream Router and can be used to split an incoming stream into multiple streams directed at different nodes or local services.

Some of this functionality can also be provided by multicast capable network protocols. In this case, the transmission and splitting of data streams can be done transparently by the underlying network layer. The ϵSOA network stack can exploit multicast support during data transmission. Like the routing support provided by the Stream Routing Layer, the multicast support should be seen as an optional component that can be used if the underlying network protocols do not support multicast.

Nevertheless, there are some use cases, where the Stream Routing Layer is required even if the underlying network supports multicast. The first use case are heterogeneous networks. The Stream Routers can be used to support multicast over subnets using different network protocols. To communicate across a network boundary, the source node can send a data stream to an intermediate node in the remote subnet. The Stream Router on the intermediate node can rely on the multicast feature of the underlying network protocol to transmit the data to the ultimate destinations.

An advanced application field for Stream Routers is the sharing of subsets of data streams. An example are two data streams that are originated by the same sensor, but at different data rates. If the data rate of one stream is a multiple of the other stream, the data stream with the lower data rate can be created by subsampling the high data rate stream, i.e., by selecting only every i'th element. Another example are streams that are sampled at the same data rate but use different filter criteria, e.g., one stream is interested in temperature readings higher than 40°C, the other in reading higher than 50°C. The filtering and subsampling can be done by splitting the original data stream and sending it to a service that performs the required processing tasks, thus creating a new data stream. The Stream Routers provide the necessary functionality to perform these tasks. At the current state, these optimizations have to be performed manually be the developer. As mentioned in Section 2.6, a direction for future research is an investigation how such optimizations can be performed automatically by an optimizer component.

3.6.5.3 Stream Id based Routing

The Stream Routing Layer presented in this section uses stream identifiers to address elements of a data stream and to store data paths in the routing tables. This addressing mechanism is well known from multicast protocols, such as IP-Multicast. The stream identifiers are used analogously to the multicast addresses used in these protocols. This addressing mechanism is beneficial in a multicast/stream sharing scenario, i.e., a scenario where multiple recipients are interested in receiving a data stream stemming from a single source. Whenever the data stream has to be split at an intermediate node, the received messages can simply be duplicated and routed to all destinations. The message itself remains unchanged throughout the transmission in the network. A target based addressing scheme creates a much higher overhead in this situation, because the intermediate node has to create a new data packet for each recipient and to add the recipients address information to the packet.

3.6.5.4 Network Configuration

The configuration of multicast environments is no trivial task. In the domain of embedded networks, we can exploit the plannability of data streams to simplify the network configuration. We chose to use an approach that strictly separates between a configuration phase and an execution phase. During the configuration phase, the routing tables on each Stream Router are configured based on the information from the application model. Because all data streams created and consumed by an application a known a priori, it is guaranteed that, after the configuration phase, every Stream Router has all the information required for handling the data streams flowing over its node.

If an application is reconfigured or a data Stream Router fails, a new configuration phase is started to adapt the Stream Routers to the new situation. Note that the transmission of data streams continues during the configuration phase. The important characteristic of the configuration phase is that it is managed by the ϵSOA middleware, which has access to global information about the network and the executed applications. The difference to the approaches for IP networks mentioned above is that the Stream Routers do not adapt their behavior autonomously. Instead, all reconfigurations are triggered by the middleware. This eliminates the need for a network protocol that handles these adaptations and allows building a very lightweight and compact implementation on the nodes in an embedded network.

The obvious drawback of this approach is an increased reaction time. Changes in the network have to be detected and reported to the middleware. After that the middleware can start a reconfiguration to adapt the Stream Routers to the new situation. A solution that is integrated into the network protocol can react much faster to such a change. But this drawback is not as severe as it appears at first glance. As mentioned in the beginning of this section, the Stream Routing Layer sits on top of a network protocol that performs the routing of packets. The majority of network changes, such as broken links or failed nodes, will be handled transparently by this network protocol. A single node failure therefore only influences the data streams using the Stream Router installed at this specific node. In many cases, such a failure will require adaptations in the communication layer anyway. The new topology might require moving services between nodes or activating redundantly available devices in order to compensate the failure. We therefore chose to accept the increased reaction time in order to gain a much more compact and lightweight implementation of the Stream Routing Protocol.

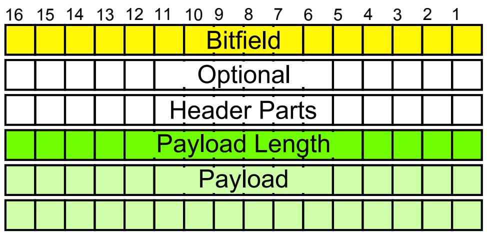


Figure 3.6: eTP Message Format

3.6.6 Packet Routing Layer

The Packet Routing Layer is optimized for ad-hoc interactions between services and administrative messages. It supports unidirectional and request-response interactions. Message exchanged via the Packet Routing Layer typically use the Reliability Module to achieve a reliable end-to-end message transfer. Due to the additional addressing information, the packet size for messages using the Packet Routing Layer is larger than the packet size for messages using the Stream Routing Layer. More details on this issue are presented in the following section.

3.6.7 Implementation

In order to create a feasible solution for resource constrained devices, the concepts presented in the previous section have to be implemented in a resource saving and efficient way. The network stack can be implemented with a single protocol that includes the functionality of the abstract network layer and both communication interfaces, the Stream Routing interface and and the Packet Routing interface. We call this protocol embedded Transport Protocol (eTP).

The ϵSOA communication protocol uses a message format that is structured as shown in Figure 3.6. A message consists of a header, comprising a bitfield, a sequence of optional header parts and a content length field, followed by the message payload. This design was chosen to support the modular design of the network stack. The bitfield is used to identify which modules in the network stack have to be activated for parsing this message. If the corresponding bit is set, the module is invoked at the receiver. Each of the modules can define extensions to the message header. These are encoded as data fields in the optional part of the header. The bitfield therefore also identifies which optional header fields are present or not.

The size of the bifield and the module identified by each bit are defined in an application domain specific profile. The profile defines the sequence of the modules in the modular stack and the size of the optional header fields created by each module. The profile ensures that all nodes implementing the profile interpret the bitfield in the same way and can parse the message header correctly and can communicate with each other. On the other hand, the protocol can still be tailored to different application domains by specifying different profiles that may contain a different number of modules and modules with different functionality. Note that the protocol profile only specifies a list of possible modules. Not all nodes in the embedded network have to use all the modules specified in the profile. The profile only ensures that each node interprets the header correctly. Which modules are installed at each node can be configured at runtime to en- or disable certain communication features depending on the application requirements.

3.6.7.1 Addressing Modules

Some modules require address information. Consider for example the Reliability Module. In order to send an acknowledgement to the sender, the receiver has to know the source node of a message. For stream based communication the Reliability Module at the receiver additionally has to know the stream a message belongs to, for message based interactions the instance and port address of the sender, respectively. The crucial observation is that these addresses are often needed by more than one module. The source address for example is also needed whenever a request/response interaction is used in order to determine the target for the response. eTP supports the sharing of address information. In eTP all addressing related information is provided by five addressing modules. The modules are located at the top of the module stack, i.e., will be parsed at the beginning for inbound messages and added as the last header components to outbound messages. The modules are:

- **Destination Node Module**: adds the destination node to the message header
- **Source Node Module**: adds the source node to the message header
- **Destination Address Module**: adds the instance and port identifier of the destination service to the header (when using message based communication)
- **Source Address Module**: adds the instance and port identifier of the source service to the header (when using message based communication)
- **Stream Id Module**: adds the stream identifier to the header (when using stream based communication)

Modules can specify dependencies to these addressing modules and other modules. A dependency specifies that the other module should also be invoked. The Reliability Module uses a dependency to the Source Node Module and the Stream Id Module (or the Source Address Module respectively). This ensures that all address information required by the Reliability Module is added to the message header. However, the address information will only be added once - even if multiple modules specify this dependency.

Besides the usage for sharing addressing information, the address modules are also used to exploit common knowledge between both ends of a communication channel. Assume a temperature sensor should deliver periodic measurements to some

Stack Position	Module Name	Header Size	Dependency
7	Encryption Module	??	-
8	Fragmentation Module	2 bytes	-
9	Reliability Module	1 byte	Source Node, Source Address, Stream Id
10	Bridge Module	-	Destination Node
11	Stream Id	2 bytes	-
12	Source Address	3 bytes	-
13	Destination Address	3 bytes	-
14	Source Node	2 bytes	-
15	Destination Node	2 bytes	-

Table 3.2: Example Profile

application logic. Because we want a reliable transmission of the sensor readings, the user/the system decides to use a TCP connection for the data transmission. Further assume that the TCP connection is not shared by multiple applications but used solely for transmitting the temperature readings. In this case, the sending and the receiving node implicitly know that all data received through the TCP connection are temperature readings. It is therefore unnecessary to transmit the stream identifier along with each data packet. eTP stores a list of such implicit information along with each communication channel. All header fields that are contained in this list are removed at the sender side and reconstructed based on the channel information on the receiver side.

We will explain these feature with some example scenarios in the following section.

3.6.7.2 Implementation Details and Example Scenarios

The examples in this section are based on the application profile shown in Table 3.2 (this profile is also used for implementing the demonstrators shown at the end of this work). The profile specifies which modules are present in the network stack, the order of these modules, and the size of the header fields of each module. The last column of the table shows the dependencies between modules. This column is not part of the profile but added to provide a better overview for the example. The dependency information is added at runtime by each module. The bitfield is not fully used by the profile. An additional number of six modules could be added to the stack. The bottommost module is the Encryption Module, followed by the other two modules presented in the previous section, i.e., the Fragmentation Module that handles the splitting of large messages, and the Reliability Module that provides reliable message delivery. The fourth module from the bottom is the Bridge Module that implements the functionality of the Abstract Network Layer, i.e., ensures communication across network boundaries. The remaining modules are addressing modules for the source and destination node of a packet, the source and destination service instances (for

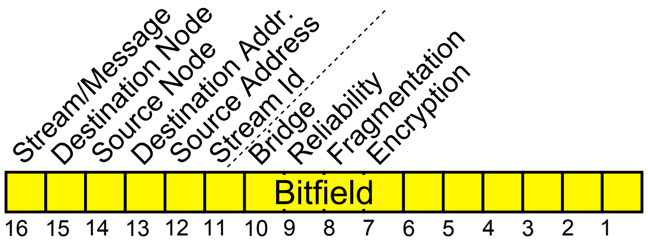


Figure 3.7: Bitfield in the eTP Header for the Example Scenario

message based interactions) and the stream identifier (for stream based interactions). The highest bit in the bitfield is reserved. It is used to identify the communication mode. If the bit is set, stream based communication is used. If not, message based communication is used.

The bitfield of the message header for this profile is shown in Figure 3.7. The highest bit determines the communication mode. If the bit is set, stream based communication is used, if the bit is not set message based communication is used. The remaining bits are used for modules. The bits 15 to 11 are assigned to the addressing modules. Bit ten is used for the Bridge Module, bit nine for the Reliability Module. The bits eight and seven are used by the Fragmentation Module and the Encryption Module respectively.

On the receiver side, the message is parsed by invoking the modules top-down. The first modules are the addressing modules, followed by the Destination Node Module. The last module is the Encryption Module. The parsing of the header fields is done with a pointer that stores the offset at which the module specific header fields are located. Initially the pointer is set to the position right behind the bitfield. If a module is activated, i.e., the corresponding bit in the bitfield is set, the module is invoked. Based on the position of the pointer, it may extract its information from the message header. After that, the pointer is incremented by the module specific offset specified in the profile. The position of the pointer is now the start of the next module header field or the start of the payload if there are no more header fields.

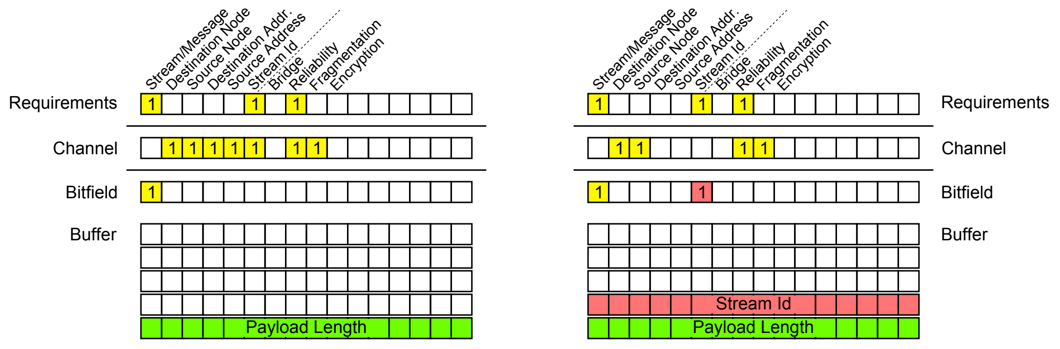
On the sender side, the modules are invoked bottom-up. Because the length of the header is not known in advance, a message is created bottom up on the sender side. First the payload is written at the end of the message buffer on the sender side. After that the header information is appended at the front. The last field of the header is always the payload length, which is therefore added at first. Next all modules are invoked in order, i.e., starting with the module at the top of the stack. In the given profile, this module is the Encryption Module followed by the Fragmentation Module and so on. The addition of header fields is performed analogously to the reading of header fields. Based on a pointer each module is assigned a part of the header to store its data fields. The only difference is that this time the pointer is decremented by the module specific header size prior to the addition of the header fields. This ensures the buffer is filled bottom-up. The last step is to add the bitfield at the start of the message header.

The invocation of modules on the receiver side is determined by the bitfield in the message header. During its invocation, each module may extract information from the message header. A module may cancel the handling of an incoming packet. In this case, no further modules are invoked and the network stack proceeds with the parsing of the next incoming packet. A typical use case is for example the detection of an invalid or malformed packet. Another example is the Fragmentation Module. If a fragment is received, this module will cancel the handling of the packet and instead store its payload in an intermediate buffer. When all fragments are available, the Fragmentation Module will recreate the original packet and pass it to the remaining modules in the stack. A module may also trigger the creation of a new outgoing packet. This packet will only be passed to all modules located below the sending module in the network stack. The outgoing packet is therefore not visible by modules higher up in the stack. The Reliability Module uses this mechanism to send acknowledgements and to re-send lost packets. Note that this communication is not visible for the Fragmentation and Encryption Modules.

Not every module is needed for every outbound packet. Which modules should be invoked is determined by a requirements bitfield that is passed to the network stack by the application. This bitfield specifies which functionality has to be provided by the communication stack. The communication stack maintains a bitfield containing the characteristics of each communication channel. Based on the requirements bitfield and the characteristics bitfield, the communication stack can determine whether a module has to be activated or not. Upon its activation, a module may set its bit in the bitfield of the outgoing message or not. If the bit is set, the module may add information to the message header. Note that setting a bit in the bitfield will also trigger the execution of the corresponding module on the receiver side. A module that has no header fields may therefore be interested in setting the bit without adding fields to the message header. A module may also decide to not set its bit in the bitfield. In this case it may not add information to the message header and the corresponding module on the receiver side is not invoked. A module that uses this functionality is the Fragmentation Module. It checks the size of outgoing packets. If a packet is small enough to fit into a single network packet, the Fragmentation Module does not set its bit in the message header. In this case the whole message is transmitted in a single packet and the Fragmentation Module on the receiver is not invoked. If the message is too large, the Fragmentation Module will create multiple small packets. It will also set its bit in the bitfield to trigger the execution of the Fragmentation Module on the receiver side, which will then reconstruct the original message based on the received fragments.

We will illustrate the functionality of the eTP protocol with some examples.

Example 1: Dedicated TCP connection for a data stream Assume a data stream requires reliable data delivery and both involved nodes are located in an IP based

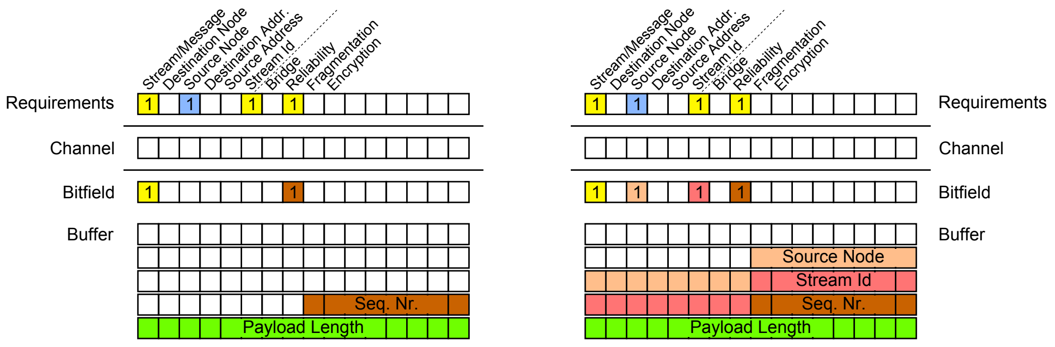


(a) Dedicated TCP Connection (b) Shared TCP Connection

Figure 3.8: Header Generation for a Data Stream Transmission

network. If both nodes support TCP and have enough spare resources, the user (or the Optimizer component in the network stack) may choose to use a dedicated TCP connection for transmitting the data stream. During the installation of the data stream, the middleware on both nodes will establish a TCP connection. Figure 3.8(a) shows the information available in the communication stack in this situation. The bitfield at the top contains the requirements specified by the application. In the example there are three requirements. Stream based communication should be used, the stream identifier should be added to the addressing part of the header and reliable transmission should be provided. The second bitfield shows the properties of the used communication channel. A TCP connection already supports fragmentation and reliable transport out-of-the box. Because the TCP connection is used only for transmitting a single data stream, the network stack on the receiving node implicitly knows all addressing information related to this stream. The third bitfield shows the bitfield that is created by the communication stack on the sender side and that is transmitted at the beginning of the message header. Below this bitfield, a buffer containing the header contents is shown. The payload length is always the last field of the message header and depicted in green in the figure. Note that this buffer is filled from bottom up, as already mentioned.

On the sender side modules are invoked top down. The application requires no encryption or fragmentation support (because all packets are known to be small enough), the Encryption and Fragmentation Modules are therefore not needed. The application does need reliable transport. From the channel bitfield, the communication stack knows that this functionality is already provided by the used communication channel. The Reliability Module is therefore not invoked. The same holds for the Stream Id Module. Because the channel provides the needed functionality, the module is not invoked. Instead the addressing information is reconstructed by the network stack on the receiver. The resulting message only contains the bitfield and the size field of the message header.



(a) Part 1: Invocation of Reliability Module (b) Part 2: Invocation of Other Modules

Figure 3.9: Header Generation for a UDP Based Data Stream Transmission

Example 2: Shared TCP connection Now assume we want to exchange two data streams between the nodes in our example. Of course it is possible to use two dedicated TCP connection for this purpose. However, in some cases it is beneficial to multiplex both streams into a single TCP connection. A typical reason for such a decision is a low amount of free memory on one of the nodes. Each TCP connection uses up resources and a resource constrained node might not be capable of supporting many TCP connections at once. The TCP connection is therefore no dedicated communication channel, but a shared communication channel. In this case, the network stack is still able to infer the source node based on the TCP connection, but not the stream identifier. Figure 3.8(b) shows this situation. The requirements are the same as in the previous example. The channel properties are different to reflect the characteristics of the shared channel. Just as in the previous example, the modules up to the Stream Id module are not invoked because they are either not required by the application or their functionality is already provided by the communication channel. The Stream Id module will be invoked because the shared channel does not provide the corresponding functionality. The module will set its bit in the bitfield of the message and add its information to the header (shown in red in the figure). For the Stream Id Module, this is a 16 bit integer value containing the stream identifier. No other modules are required, resulting in a message that contains the bitfield, the stream identifier and the payload length.

Example 3: Shared UDP channel If a reliable transport channel is not supported by the underlying network protocol, the situation is more complex. Assume we want to transmit the data stream from the first example over a UDP socket. UDP offers no reliable transport and no fragmentation support. Just like the TCP connection in the previous example, an UDP socket may be shared too. In the case of UDP, a single socket may even be used to receive data from multiple different nodes. This scenario is shown in Figure 3.9(a). The requirements are the same as in the previous

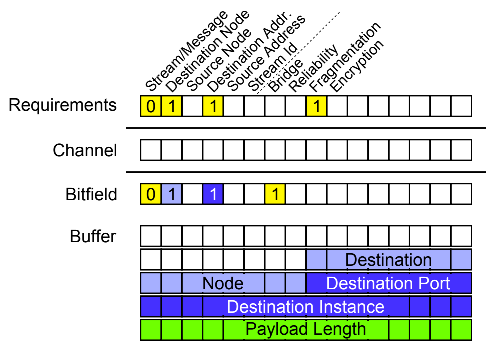


Figure 3.10: Header Generation for a UDP Based Message Transmission

examples. Ignore the blue field in the requirements for the moment. The channel possesses no useful characteristics, the corresponding bitfield is therefore empty.

We will analyze the creation of the header step by step. The first invoked module is the reliability module, because UDP provides no built-in support for reliable transmissions. The reliability module sets its bit in the header field and adds a sequence number to the message header. The reliability module on the receiver side has to know the source of the message to deliver an acknowledgement. The reliability module therefore specifies a dependency to the Stream ID module and the Source Node Module. This is done by setting the corresponding bits in the requirements bitfield. The bit for the stream id was already set so it is not changed. The bit for the Source Node Module was not set before and will therefore be set by the Reliability Module. The situation after the invocation of the Reliability Module is shown in Figure 3.9(a). The modified bitfield entry for the Source Node Module is shown in blue.

The next invoked module is the Stream ID module, which will add its information to the header and set the corresponding bit in the bitfield of the message. The same holds for the Source Node Module. The situation after the execution of all modules is shown in Figure 3.9(b). The information of the Stream Id Module is shown in red, the information of the Source Node Module in light red. The resulting message header comprises the bitfield, the source node header field, the stream id header field, a sequence number and the payload length.

Example 4: Message based Communication over an UDP Channel The last example is using a message based communication over a shared UDP channel, as shown in Figure 3.10. The application expects no reply to the sent message; it therefore only requires the destination node and the address of the destination service instance on this node. The first invoked module is the fragmentation module which is required by the application. To keep the example concise, let us assume the message is small enough to fit into a single UDP packet. The fragmentation module will therefore not add any data to the message header and will not set the corresponding bit in the message bitfield. The next module is the bridge module. The bridge module itself adds no information to the message header. Nevertheless, it sets the corresponding bit in the bitfield to ensure the bridge module is also invoked at the receiver. The bridge module has a dependency to the target node module. The corresponding bit in the header has already been set by the application, therefore no change will occur. The Destination Address Module and the Destination Node Module both add information to the message header and set the corresponding bits. Note that the Destination Address comprises two values, a 16 bit instance identifier and a 8 bit port identifier. The resulting message can now be sent to a bridge node in the network. On the bridge node, the Bridge Module will be invoked. It will check the ultimate destination of the message and send the message to the subnet the destination node is located in. This is done by cancelling the parsing of the message and by resending it through another communication channel. The bridge node on the final destination will detect that the message is targeted at its node and pass the message on to the modules higher up in the stack.

3.6.8 Summary

In this section we presented the communication layer used in the ϵSOA platform. It is based on a modular architecture and provides optimized support for the transmission of data streams. The ϵSOA communication layer allows the specification of features, such as reliability, on a per stream basis. The stack on each node can be tailored on a fine grained level by using modules. This allows creating very compact network stacks for resource constrained devices that do not require all features of the network stack. The communication layer offers built-in support for communication in heterogeneous network environments with multiple different network protocols.

There are approaches that aim at extending the IP based infrastructure used in the IT domain to embedded networks, e.g. 6LoWPAN[63] or the several approaches that aim at developing low power WLAN solutions. It is currently unclear whether these approaches will be successful or not, especially in the automation domain where bus systems play an important role. As mentioned above, the ϵSOA platform was designed to support heterogeneous networks. If a common IP infrastructure becomes available, the ϵSOA platform can be reconfigured to use this infrastructure. The bridging module is not needed in this case.

3.7 Message Parsing and Data Binding

The application fields for embedded networks are very diverse. As a consequence, the transmitted data is very different, depending on the application domain and the task of the nodes. Simple sensor devices, such as a temperature sensor, will only transmit simple data values, such as a simple numeric value. More complex devices, such as an RFID reader, will produce more complex data comprising multiple data values, e.g., an RFID code, a timestamp and a reader id. Services that are interacting with external Web services often have to deal with complex XML documents, which can have an arbitrary structure. The ϵSOA platform uses XML documents for the data exchange between services, which provide the necessary scalability for supporting the transmission of simple data values up to complex XML documents used for communicating with Web services.

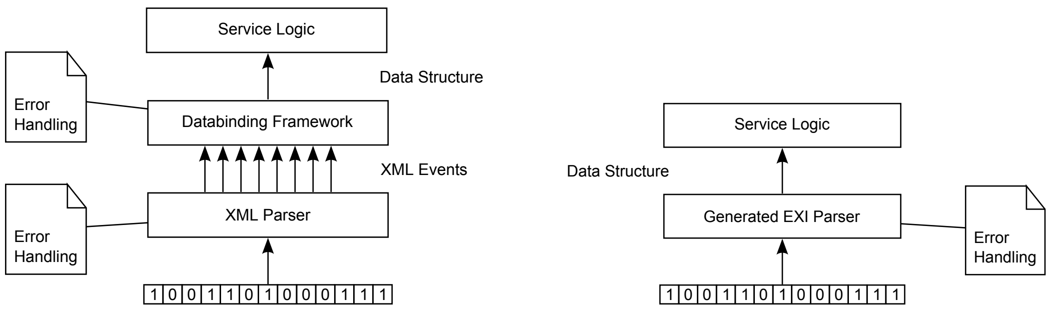
Building an efficient message parser that supports a fast and resource efficient parsing of small XML documents and at the same time supports all the flexibility provided by XML is difficult. A typical setting for embedded networks is that nodes that produce or consume only simple data values, such as sensor nodes, will also possess very limited storage and processing capacities. For these nodes, it is decisive that the size of the message parser is small and that the parsing of messages requires little CPU and memory resources. This requires a scalable message parser that is very compact and efficient for simple data formats, but can be scaled up to support arbitrary xml documents.

As mentioned in Section 3.2, the EXI format provides a compact encoding of XML data. Besides the message size, a fast and efficient parsing of messages is a prerequisite for the usage on resource constrained devices. In this section, we will show how code generation techniques can be used to create lightweight message parsers[5] for EXI, which can be executed efficiently even on severely resource constrained devices.

3.7.1 Combined Parsing and Data Binding

In nowadays solutions, the parsing of messages is often subdivided into two steps. The receiver first parses the received message in order to extract the transmitted payload. After that, the transmitted data is converted from the message data format to the data format used by the service implementation. The latter step is called data binding and typically performed by a corresponding framework. For EXI, this results in a message processing procedure as depicted in Figure 3.11(a). The received binary EXI string is parsed by a EXI XML parser. The parser creates XML events (often SAX, the Simple API for XML, or a comparable format is used), which are handed on to a data binding framework. The data binding framework creates a data structure based on the information contained in the events. This data structure is handed on to the service implementation to do the actual processing of the data.

[5] In this section, we will focus on the generation of parsers, which are used by the receiver of a message. The generation of encoders for the sender side can be performed analogously and is not explained in detail here.



(a) Separated Parsing and Databinding (b) Combined Parsing and Databinding

Figure 3.11: Message Parsing & Databinding Schemes

An alternative approach, which is based on the same message processing scheme, is the usage of a tree model instead of events for the communication between the parser and the data binding framework. In this case, the XML parser reads the whole received message at once and stores the contained information in a tree structure (e.g. a Document Object Model (DOM)[155] tree). This tree is handed over to the data binding framework and the processing continues like in the event based case.

The separation of the message parsing into two strictly separated layers increases the flexibility of the message parsing, because the layers can be exchanged individually in order to support other message formats or programming languages. But the layering also introduces a considerable overhead. The exchanged data has to be converted twice. First data is converted between the data format used in the actual message to the data format used in the XML events or XML tree structure. The second conversion is done in the data binding framework. The communication between the XML parser and the data binding framework also introduces an overhead. Even small XML documents can create a lot of XML events, because not only the contained data but also structural information, such as opening and closing tags, have to be communicated. This leads to a considerable amount of method invocations and can slow down resource constrained microcontrollers. The tree based approach does not have this drawback, but requires additional memory for storing the tree model.

Both overheads, the overhead for multiple data conversions and the communication overhead between the parser and the data binding framework, can be eliminated if a combined message parser is used that converts an incoming EXI stream directly to a data structure used by the service implementation. A corresponding message parsing architecture is shown in Figure 3.11(b). This architecture has the additional benefit that the error handling is not split up in different layers but can be realized efficiently at a single location.

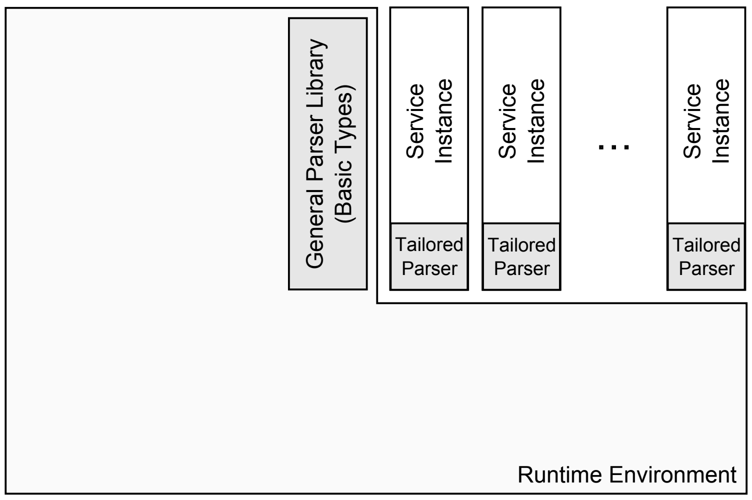
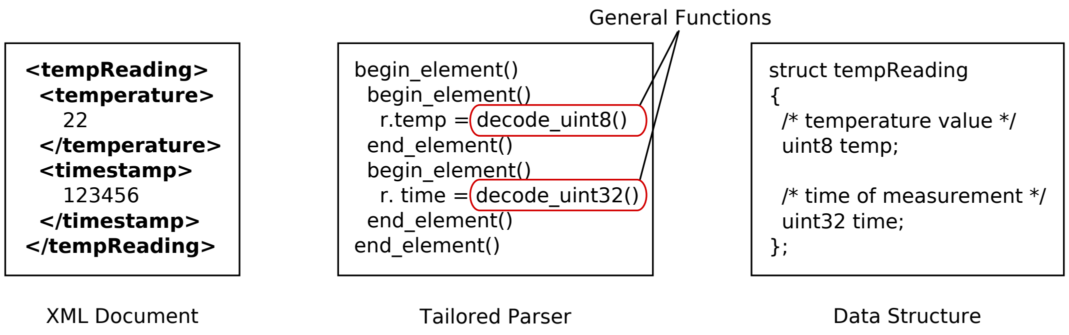


Figure 3.12: Tailored Message Parser



```
<tempReading>
  <temperature>
    22
  </temperature>
  <timestamp>
    123456
  </timestamp>
</tempReading>
```

```
begin_element()
  begin_element()
    r.temp = decode_uint8()
  end_element()
  begin_element()
    r. time = decode_uint32()
  end_element()
end_element()
```

```
struct tempReading
{
  /* temperature value */
  uint8 temp;

  /* time of measurement */
  uint32 time;
};
```

Figure 3.13: Tailored Message Parser Example

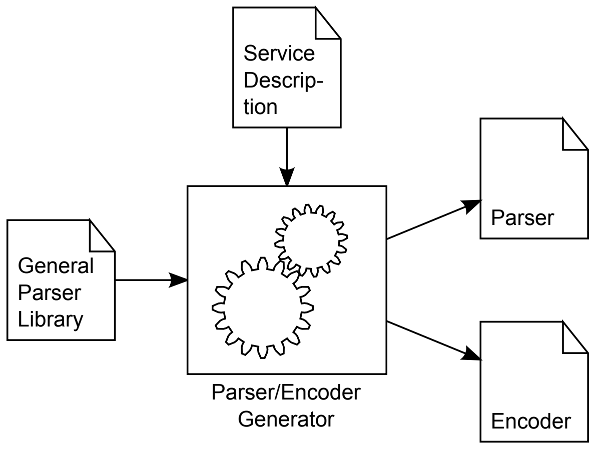


Figure 3.14: Overview of the ϵSOA Binding Generator

3.7.2 Generation of Tailored Message Parsers

To further increase the efficiency and compactness, the ϵSOA platform does not use a generic message parser, but tailored message parsers that are shipped with each service, as shown in Figure 3.12. Every node contains a general parsing library that offers support for the basic data types used by the services. The tailored parsers contain logic for the parsing of the messages received and sent by a service, including support for complex structures such as lists, optional elements, choices, etc. The tailored parsers rely on the general library for the parsing of basic types. A simple example for a sensor reading containing a temperature value and a timestamp is shown in Figure 3.13. The leftmost box in the figure shows the XML data created by the service. Note that the textual XML representation is only given for readability reasons, the message parser and encoder directly read and write binary EXI documents. The basic structure of the tailored parser is shown in pseudo code in the center of Figure 3.13. The tailored parser encodes the basic document structure, i.e., the tree of XML elements. It relies on the parsing functions contained in the general library for the extraction of simple data types, such as the unsigned integer values in the example. In the example, the read data is stored in a c-struct, other realizations for other programming languages are possible (the Java based implementation of the ϵSOA platform for example uses a Java class).

The tailored parsers are generated fully automatically based on the information contained in the service description of each service. The architecture of the parser/encoder generator is shown in Figure 3.14. Based in the service description, the generator creates structure definitions for the target programming language and tailored parser/encoder implementations. The generated code is bundled with the service implementation and loaded on the node during the service deployment.

3.7.3 Summary and Outlook

Through the combination of message parsing and data binding and the generation of tailored parser and encoder implementations, highly efficient message parsers can be realized. The parser generator used in the ϵSOA platform is optimized for deeply embedded devices with minimal resources. It does not support all features provided by EXI, especially the support for schema deviations is very limited. If these features are required, a full fledged EXI implementation has to be added to the node (which will require more resources).

The approach for code generation described in this section was optimized for generating code for documents that are exchanged in control oriented embedded networks (hence comparably simple XML documents). This is not the only application field for EXI. An EXI based encoding of XML documents is a promising approach for improving the performance of Web service based solutions in the IT domain and for speeding up the processing of SOAP messages on embedded devices. In the ϵSOA platform, messages contain solely data, the addressing information is contained in the transport protocol. SOAP messages on the other hand also contain addressing information (e.g. WS-Addressing headers or method names for rpc-style invocations). In this scenario, the message handling performance can be improved if a message dispatcher, which maps incoming messages to method invocations, is included in the parser (using a combined parsing and databinding as described in this section). Work in this direction is presented in Käbisch et al[76, 77].

3.8 Management Interface & Cross-Layer Information Exchange

Besides the communication that is needed to perform the control tasks of an embedded network, management information has to be exchanged too. The configuration of the nodes in an embedded network has to be changeable at runtime in order to adapt the network to changing application fields or changes in the structure of the embedded network. This requires an efficient and scalable management protocol that provides an easy access to the configuration parameters of the nodes and the service instances executed on these nodes. In the IT domain the Simple Network Management Protocol (SNMP)[61][146] is used on a widespread basis. To ensure a seamless integration of embedded networks and nowadays IT infrastructures, a compatible management protocol for embedded networks is desirable.

The second application field for the exchange of management information is cross-layer communication. In many cases the performance of the communication in an embedded network can be improved if the strict layering of network protocols is relaxed. If information collected by underlying layers is available at the application layer, such as the network topology, data routes or bandwidth and latency of links, the execution of applications can be optimized to better suite the characteristics of a given network. In the other direction, the transmission of messages can be improved if high level information provided by the application layer, such as data rates, is available in the network stack. This communication is not limited to an information exchange between the application layer and the layers below in the network stack. The layers inside a network stack can benefit from such an information exchange, too. We will focus on the information exchange that involves the application layer, because the development and optimization of the network protocols used in embedded networks is out of the scope of this work. Nevertheless, the presented cross-layer communication interface can be used to improve the communication inside the network stack, too.

We will first present SNMP and a design concept for a Web service based management interface in the next section and derive requirements for a management protocol for embedded networks based on these interfaces. After that we analyze existing approaches for cross-layer communication and derive requirements for the cross-layer communication interface. A comparison of these requirements shows a high overlap. We therefore developed a single solution that can be used to solve both tasks. The concepts used in the design of this solution and an efficient implementation for resource constrained nodes are shown in the main part of this section, followed by an overview of related work and future research possibilities.

3.8.0.1 Management Interface

The ϵSOA platform offers two management interfaces, which are motivated in the following paragraphs.

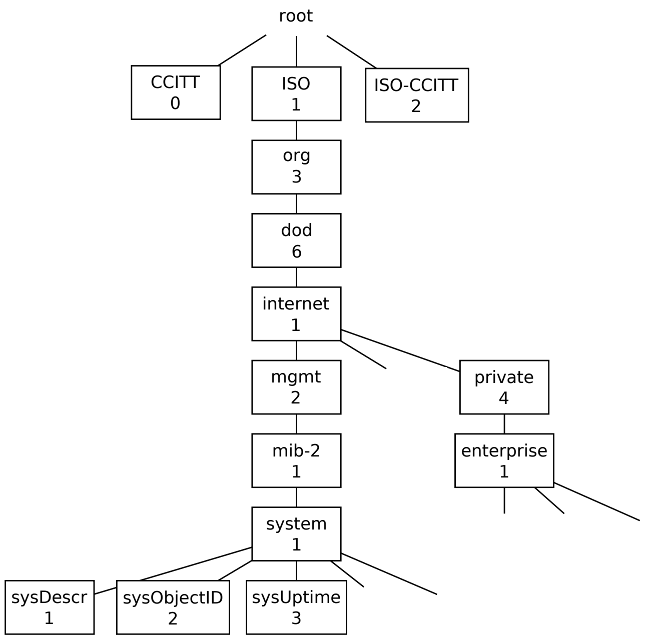


Figure 3.15: SNMP Object Identifiers

SNMP As the name implies, SNMP was initially designed for the management of network devices, such as routers in IP based networks. Because SNMP is very flexible it is nowadays not only used to configure the devices that build the infrastructure of IP networks, but also used to manage devices that are attached to these networks. In SNMP information is organized in a tree structure. SNMP uses object identifiers (oids) to address nodes in this tree. An object identifier is composed of a sequence of numeric values. Each value identifies a node in a specific level of the tree. A human readable name, the data type, a description and other related information about a node in the tree can be found in the Management Information Base (MIB). An excerpt of a tree defined in the MIB is shown in Figure 3.15. The object identifier of the system description node "sysDescr" for example is **1.3.6.1.2.1.1.1**. SNMP defines the following messages:

- Get: Used to retrieve a set of management information
- GetNext: Used to retrieve the next set of information from a table or list
- GetBulk: Used to retrieve multiple sets of information at once
- Set: Used to change/update information
- Response: Reply to one of the messages above
- Trap: May be sent from devices to a management host to report unexpected or erroneous situations

In SNMP, an object may have several instances. These instances are distinguished by appending an instance identifier to the object identifier. For objects that only have a single instance per device, such as the system description, the instance identifier is 0. Following this rule, the get request to retrieve the system description from a device has to be issued with **1.3.6.1.2.1.1.1.*0***. If an object has multiple instances (a list or a table) a single instance is identified by adding a key to the object identifier. For lists, this key is the position in the list. If a device has multiple system descriptions, the i'th description could be fetched with **1.3.6.1.2.1.1.1.*i***. For tables, the key is composed of the values contained in one or more key columns. Assume we have a table with a key column called "id". To retrieve the row identified by the entry "abc", we can issue a get request using **<oid of table>.abc**. To retrieve all values in a list or table, the object identifier without an instance identifier can be specified.

SNMP has been used for many years now and some limitations have become obvious. SNMP does not support the storage of structured data types in lists or tables. The reason is that SNMP requires that the last element(s) (and only the last elements(s)) of an object identifier are keys. It is impossible to specify multiple keys and therefore impossible to nest lists or tables. The second limitation is that SNMP specifies no operation to invoke methods on a remote node. Many complex management tasks cannot be performed by changing single values. The typical workaround in SNMP is to introduce a method call variable. If the user sets this variable to a certain value, a corresponding management function is executed by the device. This workaround is problematic because it introduces side effects that may not be obvious for an administrator. The second problem is that there is no way of passing parameters to methods. Of course there is also a workaround for this problem by storing parameters in special variables that have to be initialized prior to a method call. The solution resulting from these workarounds is very error prone and the management code issuing a sequence of such commands is hard to read and to debug.

Web Service based Management Interfaces The motivation for a SNMP based interface was to provide an interface that is compatible to established technologies in the IT domain and that allows leveraging existing management tools. Because SNMP has some limitations and does not fit well into a Web service based environment, we want to provide an additional Web service based management interface. A lot of research is done in the area of Web service based management solutions. We will only present a general overview of this work here. A more detailed description can be found in the related work paragraph at the end of this section.

The first group of related work are approaches that propose a complete redesign of management protocols based on XML and Web service technologies, such as XML, SOAP and XPath. A problem encountered during the deployment of these new management approaches is that there is a huge number of SNMP devices installed in nowadays IT infrastructures and it is very unlikely that vendors will implement new protocols for these devices. Even if all new devices would be shipped with Web service based management interfaces (what is not the case) we would still have to deal with SNMP based devices for many years. This led to the development of another group of systems: SNMP to XML gateways. These offer XML based access to SNMP based devices by transforming Web service requests/responses to SNMP requests/responses and converting binary SNMP data to XML documents and vice versa. Our solution follows the second approach.

A network management solution for embedded networks additionally has to consider the resource constraints imposed by small sized devices. The use of XML and Web service technologies for the management of IT networks is only possible because the network bandwidth and the processing power of devices in these networks has been growing continuously in the last years. This is not the case for embedded networks. There will always be application fields that require lightweight, small and cheap nodes, which will possess severely limited processing, storage and communication capabilities. In this situation, a management solution based on a comparably simple protocol like SNMP is an appealing solution. On the other hand, converged networks containing both IT systems and embedded devices require Web service based interfaces to allow a seamless integration between both domains.

Our approach was to design a protocol that uses XML and Web service technologies, but with some restrictions that allow an implementation on resource constrained devices. We use a management interface based on the REST philosophy. Management tasks require only simple request/response interactions. These can be implemented much more efficiently with a lightweight REST solution compared to the more complex SOAP based solutions. The management information is stored using an XML based information model. The selection of nodes in the XML information model is based on a XPath oriented syntax. We use a XPath dialect that is tailored for the use in management applications. It provides some additional functionality for management operations and restrictions that allow an efficient implementation on resource constrained devices. Data is stored in a binary XML format that can be handled efficiently by embedded devices and can be easily converted to plain XML if needed.

The implementation of the management interface does not store plain XML documents. We use a numeric encoding for nodes in the XML tree to reduce the storage requirements and increase the XPath evaluation performance. The crucial observation is that this encoding can be designed in a way that is backwards compatible to SNMP. Our solution is therefore a hybrid approach. It offers a SNMP based interface that allows a seamless integration into SNMP managed networks and a XML/REST based management interface for the integration in Web service/XML based management infrastructures. Both interfaces are based on the same information model and may be used simultaneously. This allows a smooth migration between SNMP and XML based management. The SNMP interface provides backwards compatibility to the existing management infrastructure, whereas the XML interface allows the integration into upcoming XML based management solutions.

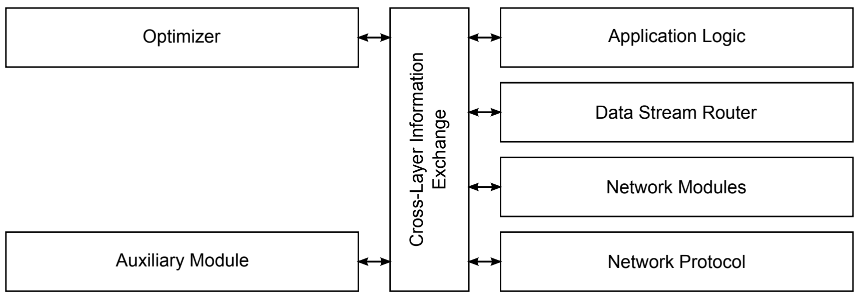


Figure 3.16: Architecture for Cross-Layer Information Exchange

Requirements An important observation is that the requirements for the SNMP and the XML/REST management interface can be easily combined. The XML solution we propose is more powerful in some areas and removes some of the limitations of SNMP. If backwards compatibility to SNMP is required, it can be achieved by using a restricted XML data model that is backwards compatible to the SNMP tree model. The same holds for the XPath based identification of objects. The object identifier in SNMP essentially is a (simple) path expression. The ePath language we propose has some additional features, such as predicates for conditional selection of nodes. Nevertheless, the basic functionality for identifying a node in the information model is identical and SNMP requests can be treated like any other path expression. A common requirement for both interfaces is that the resulting overhead for processing requests is very low to allow an implementation on resource constrained nodes.

Summing up, we identified the following requirements and design goals for the management interface:

- An information model based on XML, which is compatible to the SNMP tree model
- A XPath oriented node selection that is compatible to SNMP object identifier
- Extensions to SNMP functionality that allow invocation of methods and support nested tables
- Low storage and processing overhead to allow an implementation on resource constrained nodes
- Easy integration in Web service environments through a REST/XML interface
- Backwards compatibility to SNMP with an SNMP interface

3.8.0.2 Cross-Layer Communication Interface

The Cross-Layer Communication Interface (CLCI) is used to exchange information between the application layer and the network stack and between the different layers inside the network stack. This situation is depicted in Figure 3.16. The different components of the ϵSOA platform, such as the modular network stack, the network protocols and the application logic are connected to the CLCI. Every component can publish information and request information from other components in order to optimize its execution. The Optimizer shown in the figure is the global optimization component of the ϵSOA platform. Based on the network topology and other information provided by the lower layers of the network stack, it optimizes the execution of applications. It also publishes application level information through the CLCI to supply lower layer protocols with application level information, e.g., communication data rates.

This design is a deliberate violation of the layered design enforced by architectural standards such as the Open System Interconnection Reference Model (OSI Model)[4]. The OSI Model divides the network communication into a stack of seven layers. Each of these layers provides services to the layer above and may use services provided by the layer below. An important characteristic is that each layer only interacts with directly adjacent layers. A layer may not use services provided by a layer located more than one position down or up in the network stack. The communication between adjacent layers is limited to procedure calls and the corresponding responses. A layer therefore cannot access information provided by layers higher up in the network stack, unless this information is provided during the procedure calls.

The violation of the layered design has to be approached cautiously to avoid the creation of unmaintable code and unwanted side effects. An overview of problematic issues in non-layered network stacks is presented by Kawadia et al.[79] and Srivastava et al.[145]. On the other hand, several research projects have shown considerable performance gains if cross-layer information is exploited. The main usage of cross-layer information in the ϵSOA platform is the optimization of the execution of applications in the application level. These optimizations require information that is only visible to the lower layers of the network stack, such as neighborhood information. Our cross-layer approach does not aim at removing the layering of network protocols but instead augments the network stack with an additional communication interface that allows the retrieval of information from each layer. The CLCI can also be used to push information down the network stack. We currently use this feature to improve the monitoring in the ϵSOA platform. To detect the failure of nodes, the ϵSOA platform uses heartbeat messages that are periodically sent by each node. If it is known that a device submits data with a high periodicity, e.g., regular sensor measurements, the heartbeat is turned off and the received measurements are used to check whether the node is still available or not. This requires information about the measurement data rates, which can only be provided by the application layer. We are analyzing further tuning possibilities inside the network stack in ongoing work.

The cross-layer communication is performed using a shared information repository that can be accessed by the different components. This repository is managed by the CLCI. Because the set of running applications is not fixed, the type and structure of the shared information can change over time. This requires an extensible and reconfigurable information model. Many information accessed through the CLCI can be provided by the local node, e.g., battery levels, neighborhood information, etc. Nevertheless, some information can only be provided by combining the local information of multiple nodes in order to gain a global view on the embedded network. To allow the use of the CLCI in performance critical components such as the networks stack, the access to local information should be realized with very little overhead. The access to remote information should be handled transparently by the CLCI. The last requirement is imposed by the resource constraints on the embedded nodes. The implementation of the management repository has to be as compact as possible to minimize the required storage on the embedded devices.

Summing up, we identified the following requirements and design goals for the cross-layer communication interface:

- Extensible information model that can be changed at runtime
- Efficient access to information at the local node
- Easy access to data on remote nodes
- Low storage overhead

3.8.0.3 Solution

A comparison of the requirements for the cross-layer communication interface and the management interface shows a lot of similarities. An XML based information model solves the requirements of both interfaces w.r.t. flexibility and extensibility. Using the XPath oriented access to information, local and remote information can be addressed. With a corresponding API, a component on one node can access information published by other components on this node and information published by components on remote nodes. This provides the required access to remote and local information for the cross-layer communication. Management tasks can be realized in a similar way. The user simply specifies a path expression to select or update the information on a remote node. As we will show in the following paragraphs, it is possible to implement a XML based solution very efficiently. This allows an implementation on resource constrained nodes and provides the required performance for frequent cross-layer communication.

The information model used in our solution provides a unified view of all information available in the embedded network. It organizes the information in a hierarchical way and is the foundation for a XPath oriented query language that can be used to access and modify nodes in the tree. Its structure is flexible and can be adopted based on the application field. Please note that the tree is not materialized in the network. The represented information is still kept at individual nodes and therefore distributed throughout the network. The information tree is an abstraction that is used to organize this information. In order to interact with the information tree, meta-information about the structure, organization and type of nodes contained in the tree is required. This meta-information is kept in an external repository. We will first give an intuitive explanation of our solution based on an example.

Example An example information model is shown in Figure 3.17. The root of the information model is the *Network* node. At the current state it only contains one child node, *Nodes*, which stores a list of all nodes present in the embedded network. If additional network wide information should be added to the information model, new children can be appended. The *Nodes* element may have an arbitrary number of *Node* children. Each of these children stores information about a node in the network. A required child element is the *NodeID* element. It provides access to the node identifier in the ϵSOA platform and is needed to unambiguously identify a node in the information model. Besides the identifier, additional elements may be added to a *Node*. The type and number of these optional child elements is application domain specific and may be changed at runtime. The figure shows two additional elements that provide access to the Stream Router configuration on the node and to the configuration of the service instances installed at the node.

The *Stream Router* is implemented with a tabular data structure. The corresponding tree in the information model consists of a list of *Entry* nodes, which represent the rows in the table. Each of these rows has several columns, which are added as children to the *Entry*. To keep the example concise, we showed only three columns in the figure, *Stream ID*, *Target* and *Characteristics*. These children represent the identifier of the data stream, the target destination of a data stream and the required data stream characteristics (e.g. reliable data transfer).

The *Instances* node provides access to all service instances present at the node. Each instance is represented by an *Instance* child node. Like the *NodeID* in the *Node*, the *ID* child is used to uniquely identify an *Instance*. It is a required child of each *Instance*. The remaining children are instance specific. They can be used to provide access to status information or configuration parameter. In the example, the "Temperature Sensor" instance has a *DataRate* child that allows to retrieve and change the measurement data rate. The "Light Bulb" can be queried for the number of hours worked by retrieving the *HoursWorked* child.

The example also shows a further child element of *Node*: the *Neighbors* child. It provides access to information collected by the lower layers of the network stack. In the example it contains the identifier of the node (*NodeID*), the neighbor node (*NeighborID*), the *Bandwidth* and the approximated *Distance*, which is calculated based on the received signal strength of the radio signal. Like the *StreamRouter*, the *Neighbors* node is organized as table. *Neighbors* can be used locally on a node to exchange information between the different layers of the network stack. If the neighbor information from multiple nodes is collected, the network topology can be reconstructed. This information is used during application installation to optimize the execution of application based on the network characteristics.

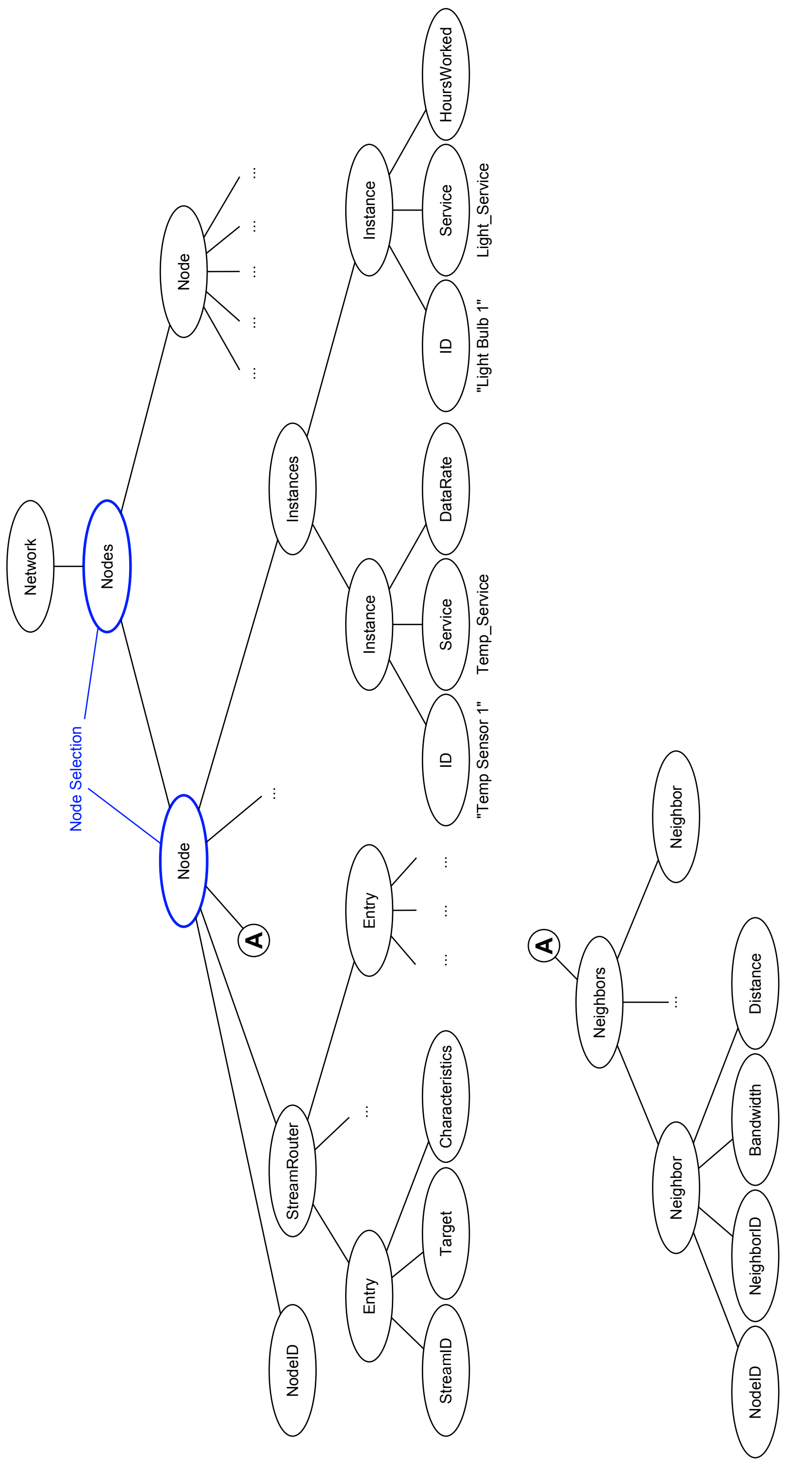


Figure 3.17: Example Information Model

Structure The tree structure used for organizing the management information uses two conventions. Lists are implemented following a convention we already used in the example for the list of nodes. A list of n item is always composed of a single parent element (*Nodes* in the example) that has n child elements (*Node* in the example). The parent may not have any other elements besides these children.

Tables are realized as lists of tuples. A single parent element stores a child for each row in the table. Each row possesses a child element for each column in the table. We already used this convention in the example for modeling the *StreamRouter*. The *StreamRouter* possesses a *Entry* child element for each row of the routing table. Each *Entry* possesses a child for each column in the table.

These conventions are used for two reasons. First they guarantee backwards compatibility to SNMP, which can only express information models following these conventions. Second, they allow a more compact representation of the tree. We will present more details on both issues in the following sections.

Path Expression Language Users or programs can access the information tree by selecting a node of the tree and issuing a data retrieval or update request. A node is uniquely identified by the path from the root to this node. We use a XPath oriented language to notate such a path. XPath is well known in the IT domain and should therefore provide an intuitive query interface for many users. We use a subset of XPath that can be implemented efficiently on resource constrained nodes. We call this subset embedded path expression language (ePath).

Like XPath, ePath uses localization steps comprising a navigation axis, a node test and optional predicates. An expression consists of a sequence of localization steps separated by /. The root of the tree is identified by the /. Multiple nodes of a tree can match a given path expression. If this is the case, a list containing all matches is returned, or an update of all matches is performed respectively. Consider the following example path expression

```
        /Network/Nodes/Node[2]
```

The expression consists of three localization steps. First the root node of the tree is selected and a node test is performed to check whether the node is "Network". The second localization step, /Nodes, selects all children of type "Nodes" from "Network". Like in XPath the default axis is the child axis and can be omitted. The third localization step selects all "Node" children of "Nodes". This step additionally uses a predicate. Predicated are notated in square brackets and appended to the localization step. A simple numeric predicate i selects the i'th child element that passed the node test. In the example we selected the second "Node" child. It is also possible to use boolean expressions as predicates. For example

```
        /Network/Nodes/Node[NodeID = 4]
```

selects the node with node identifier 4. Boolean expressions can be built based on constants and direct children of a node. In the example we compare the value stored in the child "NodeID" with the constant 4. The limitation to direct children is more restrictive than XPath. XPath allows arbitrary XPath expressions inside the predicate. We chose this limitation to get a more compact implementation of the query execution engine on resource constrained nodes.

To retrieve the data rate of the temperature sensor from the information tree in the initial example, we can use the following expression:

```
/Network/Nodes/Node[1]/Instances/
                    Instance[ID = ``Temp Sensor'']/DataRate
```

The path expressions are also used for exchanging cross-layer information on a node. Many requests will access information that is available at the node the request is created on. The home operator $ can be used as a shorthand expression. The definition of $ is:

```
$ ::= /Network/Nodes/Node[x]
```

where x is the identifier of the node, the request is issued at. If the temperature sensor in the example above is located on the local node, the request can be rewritten to

```
$/Instances/Instance[ID = ``Temp Sensor'']/DataRate
```

An important requirement for the network management is the support of method invocations. ePath supports this features with a special axis, the method axis #. The method axis can be used to invoke a method provided by the element specified with the preceding localization step. Assume the “Stream Router” has a method named “clear” that can be used to wipe the internal routing table. To invoke this method on the local node, the following expression can be used

```
$/StreamRouter/#clear()
```

Note that the “clear” method is not a built-in method from the query language. It is a method offered by the “StreamRouter” and has to be implemented outside of the query language. These methods are therefore fundamentally different from XPath functions, which are a feature of the query language. ePath defines two standard methods, “set()” and “get()”. The former is used to change the value of a node, the latter is used to retrieve the content of the current node (for leaf nodes) or the subtree rooted at this node (for inner nodes). Note that “get()” is the default method that is called whenever no other method is specified. The data rate selection above is therefore equivalent to

```
$/Instances/Instance[ID = ``Temp Sensor'']/DataRate/#get()
```

The “get()” method has an optional parameter n that allows to limit the depth of the returned tree. An invocation of “get(1)” will only return the direct children of a node.

Methods can have an arbitrary number of arguments. These arguments are passed on to the externally implemented method. The “set” method accepts one parameter, the new data value.

The wildcard node test can be used to select all children of a specific node. To retrieve all values belonging to the StreamID column of the Stream Router, the following expression can be used

```
        $/StreamRouter/*/StreamID
```

The return value of this function is a list containing the corresponding entries. Please note that there is a difference between the following two expressions:

```
        $/StreamRouter
```

and

```
        $/StreamRouter/*
```

The former selects the subtree rooted at the Stream Router node. The return value is a single tree. The latter selects all entries of the Stream Router table. The return value is a list of trees. This list contains all children of the Stream Router node.

Implementation Fundamentals A textual representation of the nodes in the information tree is not feasible for resource constrained nodes. Even if dictionary tables are used and each node only refers to an entry in the dictionary, the resulting storage requirements are too large. In the example, a dictionary containing only the elements used in the subtree of the "StreamRouter" would already require more than 60 bytes of storage. The demand for the more complex "Instances" subtree will be even higher. The required capacity for storing the information tree from the example is therefore a few hundreds of bytes. The example is a comparably simple tree for a node executing two service instances. The storage requirements quickly increase with every additional instance. The required storage capacity is not feasible, given the RAM constraints in the order of 10 kBytes (the nodes we used in the demonstrator have a total of 11k RAM).

To reduce the storage requirements, we use a numeric encoding of the tree nodes. Each node is assigned a number that is unique at the child level, i.e., all children of a node must have distinct numbers. Instead of storing the name of the nodes in the tree, we only store this number. The resulting storage requirements are in the order of some tens of bytes, what is a magnitude smaller than the dictionary based solution. The mapping between node names and numbers is stored in a Meta-Information Repository, along with the data type definitions and the structure of the information tree. The assignment of the numbers has to be coordinated to ensure the interoperability between components from different vendors. A corresponding infrastructure has already been established for the Management Information Base (MIB) in SNMP. Besides a general set of management information specified in standards such as the MIB-II[57], manufacturers can request MIB identifiers and publish management information for their devices in a subtree under this identifier. The MIB is flexible enough to include the management information required for embedded networks. General management information can be added in a specific subtree of the MIB. Device specific information can be added by the manufacturer using the extensibility features included in the MIB.

Meta-Information Repository The Meta-Information Repository describes the structure of the information tree. Each entry in the repository describes a node in the tree and contains the following information:

- Namespace: namespace of the identifier
- Identifier: identifier of this entry
- Parent: identifier of the parent node
- Position: unique number (unique on child level)
- Type: type of the node
- Description: textual description of the entry

The namespace entry is explained in detail in the next section. It can be ignored up until then.

The Identifier uniquely identifies each entry and is used to create references between elements in the Meta-Information Repository.

The Parent field is used to specify the identifier of the parent node. This field is mandatory for every element but the root.

For performance reasons, the implementation of the information model uses numeric values instead of names for each node. The Position field provides these numbers. Positions have to be unique on the child level, i.e., two children of a node must have distinct Position values. The Position field is required for every entry, unless the item belongs to a list. In this case, the Position is determined by the position of the item in the list.

There are three classes of datatypes: simple types, Objects, Subtypes, Lists and Methods.

Simple types are basic datatypes that can be used to store values in the leafs of the tree. We support all simple XML Schema datatypes.

Objects are compound data types. An Object can be composed of simple types and other Objects.

Subtypes are a convenience construct for specifying multiple objects that share common parts. A Subtype inherits all children of the supertype. A Subtype may specify additional children. If Subtypes are used, the Position numbers have to be unique for the union of sub- and supertype children, i.e., a subtype child may not reuse a Position number already occupied by a superclass child.

Lists are specialized Objects. Children of a List have no fixed Position number. Instead their position is determined at runtime based on the insertion order in the list. Children of a List containing n elements are always numbered in increasing order starting by one. It is possible to add nodes of different types to the same List.

Namespace	Identifier	Parent	Position	Type
default	Network	-	1	Object
default	Nodes	Network	1	List
default	Node	Nodes	-	Object
default	NodeID	Node	1	INT
default	Stream Router	Node	1	List
default	Instances	Node	2	List
default	Entry	StreamRouter	-	Object
default	StreamID	Entry	1	INT
default	Target	Entry	2	INT
default	Characteristics	Entry	3	INT
default	clear	StreamRouter	3	Method
default	Instance	Instances	-	Object
default	ID	Instance	1	INT
default	Service	Instance	2	INT
TempService	TempService	default:Instance	-	Subtype
TempService	DataRate	TempService	3	INT
LightService	LightService	default:Instance	-	Subtype
LightService	HoursWorked	LightService	3	INT

Table 3.3: Meta-Information Repository for the Example Information Model

Methods can be added to Objects (and the derived types Subtype and List). A method can have an arbitrary number of parameters. A parameter can be any simple type and is specified as child of the Method. Methods always have a return value that specifies whether the invocation was successful or not. The Position number of Methods has to be unique on the method level, i.e., no two methods may have the same Position number. Positions 1 and 2 are reserved for the “get()” and “set()” methods.

Table 3.3 shows the Meta-Information Repository for the running example (the “Neighbors” subtree is omitted to keep the example concise). The root node is the Object “Network”. It possesses a single child element “Nodes”, which is a List. Possible entries of the list are all children of “Nodes”, in this case “Node” Objects. Note that “Node” has no Position value because it is the child element of a List. A node has two children. The first child is the “Stream Router” identified by a Position value of 1, the second one the “Instances” List, identified by a Position value of 2. The “Stream Router” is a List of “Entry” Objects. Entries are composed of three integer values: a “Stream ID” (Position 1), a “Target”(Position 2) and “Characteristics” (Position 3). The “Stream Router” also specifies a Method named “clear()”. This method has no parameters and therefore no children. The second child of “Node” is the “Instances” List. As seen in the example, “Instances” can contain different service instances with different structures. This is modeled by specifying the common parts of these structures in the supertype “Instance”. The common parts are the “ID” and “Service” simple types. Based on this supertype, two subtypes are defined. The “Temp_Service” subtype extends the supertype with an additional field “DataRate”, the “LightService” with an additional field “HoursWorked”.

```
<mir:entry xmlns:mir="http://www3.in.tum.de/mir">
  <!-- define new namespace for service -->
  <namespace>TempService</namespace>
  <!-- this entry defines the TempService inner node -->
  <identifier>TempService</identifier>
  <!-- TempService is a subtype of Instance -->
  <parent>Instance</parent>
  <!-- TempService is only used in lists, no Position needed -->
  <position></position>
  <!-- TempService is a subtype -->
  <type>Subtype</type>
</mir:entry>
```

Listing 3.10: Meta Information for TempService

```
<mir:entry xmlns:mir="http://www3.in.tum.de/mir">
  <!-- namespace is inherited -->
  <!-- DataRate defintion -->
  <identifier>DataRate</identifier>
  <!-- additional field for TempService -->
  <parent>TempService</parent>
  <!-- numeric identifier -->
  <position>3</position>
  <!-- data rate in measurements per minute -->
  <type>short</type>
  <!-- data rate is managed by the device and may only be queried -->
  <access>read-only</access>
  <!-- has to be implemented by every device -->
  <status>mandatory</status>
</mir:entry>
```

Listing 3.11: Meta Information for DataRate

As indicated in the example, each service instance can extend the Meta-Information Repository with a tailored management interface. To support an easy integration of new devices into an existing embedded network it is desirable that these extensions are stored directly on the node. If a new node enters the network, the management meta-information can be retrieved automatically from the node and added to the repository. Information in the MIB of SNMP is specified according to the Structure of Management Information Version 2 (SMIv2)[60] specification. SMIv2 specifies a textual data format to represent meta information. The resulting descriptions are too large to be stored efficiently on resource constrained devices. We designed an XML format to store entries in the Meta-Information Repository. It contains all fields shown in Table 3.3. The format additionally contains fields for storing the information contained in SMIv2 documents, such as access rights and status flags. These fields have the same purpose as the corresponding fields in SMIv2 and are not presented in detail here. The XML-Schema definition for this format makes heavy use of enumerations. Enumerations can be encoded very efficiently with EXI and the resulting document sizes are small enough to fit on embedded devices. Example documents for the temperature service are shown in Listing 3.10 and Listing 3.11, the corresponding XML-Schema definition can be found in Appendix B.4. Both documents use the SMIv2 field *access* to specify the valid management operations on the node. The second document additionally uses the SMIv2 field *status* to specify that the corresponding management entry has to be implemented by all devices. The EXI encoded documents from the examples both have a size of 28 bytes.

Namespaces The uniqueness requirement for identifiers can be relaxed. This is done through the creation of namespaces. A namespace can be created at any node and includes the node and all nodes belonging to the subtree rooted at this node. A namespace behaves like an implicit prefix that is added to any node identifier belonging to the namespace. Because of this prefix, nodes from different namespaces can be distinguished even if they use the same identifier. Namespaces are stored in the Meta-Information Repository. The first column of Table 3.3 specifies the namespace of an element. Namespaces may be omitted in the reference to the parent node in the meta-information repository, if both nodes belong to the same namespace. In all other cases, they are mandatory.

Note that, despite the introduction of namespaces, the path to a node in the information model is still unique if the numeric values are used. The path is determined by the Position numbers which are by definition unique at the child level. A sequence of these numbers therefore unambiguously identifies a node in the information tree.

To keep the path expression language concise, namespace identifier may be omitted if the node identifier is unambiguous. This is the case in the vast majority of path expressions. Ambiguities occur whenever a node has two or more children that all use the same identifier but from different namespaces. The "Instances" node in the example is a possible candidate for ambiguities. Assume two developers modeled two instances, each using a different namespace (what is a good design decision to avoid name clashes). Assume both instances have a "DataRate" node. The following ePath expression is ambiguous:

```
        $/Instances/Instance[ID = ''Temp Sensor'']/DataRate
```

Based on the information contained in the Meta-Information repository, it is unclear to which namespace the "DataRate" field belongs. Both instances are subtypes of "Instance" and both use a "DataRate" field but from different namespaces. This information could only be retrieved by actually executing the path expression and checking the type of the instance with ID "Temp Sensor". An additional complexity is that ambiguous expressions cannot be transformed to a numeric encoding, because the different candidate identifiers may use different Position values. Ambiguous path expressions are therefore rejected by the execution engine and have to be reformulated by the user. This can be done by adding the namespace identifier:

```
        $/Instances/Instance[ID = ''Temp Sensor'']/TempService:DataRate
```

In some cases ambiguities can be removed by using the specific subtype instead of the generic supertype. The above path expression is equivalent to:

```
$/Instances/TempService[ID = ''Temp Sensor'']/DataRate
```

This path is unambiguous, because the subtype TempService specifies only a single DataRate entry.

Implementation As mentioned in the previous sections, the implementation uses numeric node identifiers instead of strings for performance reasons. The Information tree offers two interfaces, the ePath API that is used to evaluate textual ePath expressions and the binary API that can be used to evaluate numeric path expressions. The former is typically used to perform network management tasks using the XML/REST interface. It offers a human readable syntax and provides an intuitive interface for the end user. In order to evaluate a path expression, it is first transformed to a binary path expression. This transformation overhead can be avoided by using the numeric interface. It is typically used for the cross-layer information exchange between components or management tasks using the SNMP interface. The path expressions used by these components will not change at runtime. They can be transformed once during development and the more efficient numeric representation is used at runtime. The query evaluation is organized in five steps, which are explained in the following paragraphs

If the ePath interface is used, the first step performed in the execution engine is the *transformation* of ePath expression into the numerical notation. This can be done easily based on the information in the Meta-Information Repository and is not presented in detail here. If the numeric interface is used, this step is omitted. Assume the following ePath expression is issued at node 1, which also maintains the Meta-Information Repository:

```
/Network/Nodes/Node[Logical ID = 4]/Instances/
        Instance[ID = ''Temp Sensor'']/Temp_Service:DataRate
```

This ePath is rewritten by the evaluation engine on node 1 to the following numeric path expressions:

```
/1/1/*[Logical ID = 4]/3/*[1 = ''Temp Sensor'']/3
```

As stated in the previous sections, nodes contained in a list have no fixed Position entry. The evaluation engine replaces the corresponding entries in the path with the wildcard operator $*$ to ensure the whole list of children is traversed.

The second step is the *selection of target nodes* of an expression. The topmost three layers of the information tree in the example specify a node in the embedded network. The execution engine splits the path expression at this point. The first part, including any predicates and the home operator $, is evaluated. The result of this evaluation is the identifier of a node (or a list of identifiers). Note that because the position of nodes in the node list is not deterministic, a specific node can only be accessed by specifying a predicate involving the NodeID. If multiple nodes are affected by a path expression, the following steps are performed for each of these nodes. In the example, the path expression is targeted at a single node identified by the following expression:

```
        /1/1/*[Logical ID = 4]
```

The target node therefore is the node with id 4.

The third step is the *transmission* of the path expression to the target nodes. This step can be omitted if the only target is the local node.

After the completion of the previous steps, the fourth step is the *evaluation* of the path expression at the target node. The purpose of the information tree is to provide access to information that is stored in the various components installed at a node, such as service instances, the Stream Router or the network stack. A design goal was to avoid storing redundant information. Instead of duplicating the information in the information tree, we use callback functions that provide access to the information installed in the corresponding component. This ensures that the returned data is always up-to-date and minimizes the overall storage consumption. A callback function has to be implemented by every component that wants to publish information in the information tree. The callback function has the following syntax:

$$callback\_function(methodType, remainingPath, < parameter >)$$

The individual components are:

- **methodType**: defines which method should be invoked. Possible values are the predefined functions get() and set(), or user defined functions.
- **remainingPath**: a part of the original path expression that has to be evaluated by the method (this field is explained in more detail later on)
- **<Parameter>**: parameters for the method call. Number and type depend on the invoked method.

The evaluation of the path is done by subsequently processing the individual parts of the path expression. The evaluation is done by maintaining a list of nodes n that match the path expression. Initially, this list contains only the root node. Let p be the path element that should be evaluated. The engine evaluates the path expression for each entry in n_i and thereby distinguishes four cases:
Case 1: *Evaluation Finished*
If p is the last path element, the evaluation of the expression is finished. If the user specified a method call, the corresponding method is invoked on n_i. If the user did not specify a method call (or used the “get()” method), the subtree rooted at n_i has to be returned (if n_i is a leaf, this tree consists only of a single node). This

is done by calling each callback function located in the subtree of n_i and combining the results according to the structure of the information tree.

Case 2: *Inner Node Found*

If p is not the last path element and n_i is an inner node, n_i is removed from the list n and replaced with the child node(s) identified by p. If a wildcard expression is used, all children of a node are appended to n. If a predicate is specified in p, the list is filtered accordingly.

Case 3: *Callback Found*

If p is not the last path element and n_i is a method call, the method is invoked with the remaining part of the path expression as parameter. This is a very convenient way for providing access to information stored in lists or tables. The Stream Router for example installs the callback function at the "StreamRouter" node of the information tree. An access to

```
        .../StreamRouter/Entry[4]/StreamID = .../1/4/1
```

will result in a method call to the Stream Router with a remaining path of 4/1. In this case, the first element identifies the row in the Stream Router table, the second element the column.

Case 4: *Error*

In any other case, or if an error is encounter during the invocation of the callback method, the path expression is invalid and a corresponding error is returned to the user.

The fifth and last step is the transmission of the result. If multiple nodes were involved in the execution of a path expression, the results are combined in this step. Results are encoded as XML trees using the binary XML format EXI.

Web service based interface The Web service based management interface has been implemented for our demonstrator platform. It follows the REST philosophy. The ePath expression can be specified directly in the URL of a http get/post request. Return values are delivered as XML document in the http response. This interface is used by the development and management tools in the ϵSOA platform to adjust the configuration of the nodes and services in the embedded network. The Web service based interface is offered by a lightweight bridge. The bridge is required because not all embedded networks support IP based communication, especially the TCP/IP communication required by http is often not supported. The bridge transforms incoming http requests into corresponding requests in the ϵSOA platform and vice versa. The bridge also performs the mapping of ePath expressions to numeric path expressions. Note that the conversions only affect the message format. Because the internal data model uses a XML based data format the actual payload of the exchanged messages is not changed by the bridge. If the embedded network supports TCP/IP communication, the bridge is not required. However, a bridge might still be useful in this case for providing access control or maintaining the Meta-Information Repository.

Compatibility with SNMP The tree based information model and the numeric path expressions were designed to be backwards compatible to SNMP. The information model used in the ϵSOA platform is more expressive than the SNMP information model w.r.t. tables and lists. In SNMP, a table may only contain simple data values. In the ϵSOA platform a table may contain complex values, too. This limitation is inherent to SNMP and well known by SNMP experts. To circumvent this problem, special table designs have been developed for SNMP that allow the modeling of tables containing complex data types. The ϵSOA information model does not enforce these limitations. If a SNMP based interface should be offered, these considerations have to be taken into account during the development of the information model. Otherwise the corresponding subtree of the information model can only be accessed with the Web service based interface. SNMP uses a slightly different syntax compared to ePath to address elements in lists and tables. Both use keys to identify entries. In SNMP, keys are always appended at the end of an object identifier. In ePath, keys are specified with predicates and may occur at any position in the path expression[6]. These two modes can be supported easily with a switch in the query engine.

The SNMP based interface in the ϵSOA platform is not fully implemented yet. The missing part is a bridge component that offers three functions: (1) a conversion between the UDP based SNMP protocol and the message format used in the ϵSOA platform, which is not necessarily based on IP, and (2) a transformation between the XML message format used in the ϵSOA platform and the ASN.1 format used by SNMP. Note that the latter transformation is straightforward, because the XML structure is identical to the tree structure specified in the MIB/Meta-Information Repository.

The third task of the bridge is the mapping between the information models used in both domains. The syntax of the numeric path expressions is compatible to the syntax used in SNMP. The information model specified in the Meta-Information Repository can therefore be mapped to the information specified in the MIB repository in SNMP. The only prerequisite is the addition of a prefix that embeds the ϵSOA information model in the hierarchy specified by the MIB, which can be done by the bridge. The bridge can also generate a MIB module definition containing all information of the Meta-Information Repository, which can then be loaded by SNMP tools.

A possible extension of the bridge is to support the efficient bulk retrieval of management information. SNMP uses an iterator model to traverse tables and lists. A requester can issue a series of "getNext" requests to retrieve one entry after the other. This results in a very high number of exchanged messages and a high communication overhead. The bridge could be configured to prefetch specific tables upon the arrival of the first "getNext" request. This can be done by simply fetching the whole subtree defined by the table. The series of "getNext" requests can then be answered directly from this cache. SNMPv2 introduces a new bulk oriented get

[6]This difference is the reason why ePath supports nested tables, whereas SNMP does not.

operation to improve the retrieval of multiple data items at once. If such a request is issued, the bridge could automatically activate the caching for the corresponding subtree.

Cross-Layer Information Exchange The second application field of the information tree - besides the management interfaces explained in the previous sections - is the cross-layer information exchange. The tree based information model is flexible enough to allow the addition of new cross-layer information at runtime. If a new component is installed at a node, or an existing component is replaced, the information model can be adapted accordingly.

At the current implementation state, the cross-layer interface in the ϵSOA platform is used primarily for topology discovery. Each node maintains a list of neighbor nodes. This list contains information about the signal strength and related metrics of received messages, which is supplied by the physical layer of the network stack. The list is also used to track the number of corrupted or lost packets and link related metrics such as bandwidth, utilization, etc. This information is added by the used network and transport protocol. The signal strength is used to approximate the distance between nodes and to layout nodes in the graphical user interface. The other metrics are stored in the System Model in the Abstract Network Layer and are used to optimize the execution of applications. Not all metrics can be supplied by all protocols. The extensibility features of the cross-layer interface can be used to dynamically add or remove information based on the capabilities of the used protocols. In order to create the network topology and the Abstract Network Layer view, all neighbor tables have to be collected at a single node.

The information model provides the application developer with an easy to use interface to access data stored on remote nodes. By supplying suitable node selection criteria at the first levels of the ePath expression, data from one or more remote nodes can be fetched. To retrieve a unified table containing the information of all neighbor tables, the ePath expression

```
        /Network/Nodes/*/Neighbors/*
```

can be used. The result of this request is a single big table containing the entries of all neighbor tables. The required network communication is handled transparently by the system.

The ϵSOA platform continuously monitors the availability of nodes to detect node failures. The monitoring is realized with a periodic heartbeat. If no signal is received for a configurable period of time, a corresponding failure notification is issued by neighboring nodes[7]. The heartbeat is not needed if nodes emit measurements with a high frequency. In this case, the packets used for sending the measurements can be used to detect whether a node is running or not. Information about the data rates of data streams and measurement frequencies are provided through the cross-layer

[7]A node failure is only announced if no other node receives the alive signal to distinguish between broken links and a complete node failure.

interface by the application layer. The ϵSOA middleware automatically adjusts the frequency of heartbeats based on this information to avoid unnecessary message transmissions.

Related Work A good overview of different network management approaches can be found in the book of Martin-Flatin[102]. An overview about recent developments in the area of Internet Management is provided by Schönwälder et al[133]. We will focus on projects that either use SNMP, XML based interfaces or are specifically targeted at embedded networks.

Management Protocols in the IT domain SNMP is not the only network management protocol available in the IT domain. An alternative is Universal Plug and Play (UPnP). UPnP defines a set of protocols that ease the installation of components in home and enterprise networks. UPnP provides protocols for address assignment (DHCP based or based on AutoIP[62]), service discovery, service description, control operations on the devices, event notifications from the devices and a presentation layer based on Web technologies. The benefit of UPnP is a large set of functionality that covers many important aspects of system management. The drawback is the complexity required to implement this functionality. We could not find an UPnP implementation that is small enough to fit on small embedded devices such as the TMote platform. In order to be usable in the area of embedded networks, UPnP has to be adapted. The addressing functionality is tailored to IP based communication which may not be used in embedded networks. The SOAP based communication and the XML based description format may require some changes to fit on resource constrained nodes. The presentation layer based on a Web browser and http will be hard to realize on embedded devices due to its size. It is possible to implement a Web server on the TMote platform, however the implementation of the TCP/IP stack, the http protocol and the Web service use up almost all resources available on the device, leaving no room for the implementation of the actual applications. Instead of redesigning UPnP in all these areas we chose to start with a more lightweight protocol, SNMP, that is focused on the implementation of the management tasks alone and creates a much lower implementation footprint.

Due to the same reasons we also restrained from building a management solution based on the Devices Profile for Web Services (DPWS). DPWS aims at providing the functionality of Web service technologies on resource constrained embedded devices. DPWS offers functionality comparable to UPnP, with the addition of a service registry that provides a centralized overview of all available services in the embedded network. DPWS uses stripped down versions of the WS-* technologies to allow an implementation on small devices, however the resulting resource usage is still very high compared to SNMP. A lot of this complexity can be attributed to the SOAP/XML based communication. In our opinion, a REST based communication fits better into the domain of resource constrained devices. We therefore decided to implement Web service based management with a REST based interface instead of DPWS. But this decision is not fixed. There are ongoing efforts to improve the performance of DPWS[48]. If a resource efficient DPWS implementation becomes available, the REST based frontend can be changed to a SOAP based frontend using DPWS.

The Web-Based Enterprise Management (WBEM) consortium defines a set of technologies for Web based management solutions. The core technologies are the (meta)schema definition CIM, a mapping of CIM to xml called xmlCIM, and protocol bindings for the CIM operations for http and WS-Management. CIM is based on a metamodel that uses an object oriented design. The available metamodel entities are: classes, properties (state), methods (behavior), qualifiers (metadata), etc. The Common Information Model (CIM)[27] specifies three model layers. The Core Model contains a basic vocabulary for describing managed systems. This model is extended by Common Models that define further classes for technology independent application areas, e.g., network management. The Core and Common Model are defined in the CIM Schema specification[32]. The Common Model can be further refined by technology specific extension specified in Extension Schemata. The CIM model itself is not bound to a particular representation. A suitable XML based format is specified in xmlCIM[30], the corresponding DTD definition in [29]. CIM supports two kinds of operations. Extrinsic operations are standard methods defined in a CIM schema. Intrinsic operations can be used to analyze and modify the class model. They provide an interface comparable to reflection APIs in object oriented programming languages. The interaction with a CIM based system is possible using different bindings, such as http[28] or WS-Management[31].

There are other management approaches that are also based on a stack of standardized XML technologies. Hong et al. propose a management system based on Web servers running on the embedded devices [151]. In this system, management information is exchanged in a XML format and transported with http calls. The authors also propose a SOAP based interface for implementing management operations [17]. To achieve an integration with SNMP based systems, the authors propose a XML/SNMP gateway [74].

Juniper Networks offer an interface called JUNOScript on their routing platforms[75]. JUNOScript allows client applications to communicate with XML based messages with a Juniper router. The message format is specified in DTDs and XML Schema documents along with a documentation describing the semantics of the message elements. A white paper from Juniper Networks is also available motivating the usage of XML for network management[134].

The Web-based Integrated Management Architecture (WIMA)[102] allows an integration of multiple management data models, including SMI and CIM. The integration of the different models can be performed using either a model-level or a metamodel-level mapping. Using a model-level mapping each module specified in SMI is mapped to a dedicated DTD. Using metamodel mapping, a generic DTD is used to represent all modules in the MIB. The benefit of model-level mapping is that the resulting XML documents are easy to read because they reflect the structure used inside the MIB. The drawback is that there may be multiple possible mappings between both data models. The mapping process cannot be automated due to this reason. The opposite is true for metamodel-level mapping. The generic XML model can be created fully automatically, but is hard to read.

Klie and Strauß[147] propose an approach to automatically convert XMI based MIB entries to XML-Schema definitions. This approach is comparable to the model-level mapping used in WIMA. The authors also propose some optimizations to achieve a better readability of the resulting documents, such as reducing the depth of the resulting document by flattening the tree structure. Based on this mapping, the authors designed a gateway that allows an integration of SNMP managed systems into a XML management infrastructure[83].

Pras et al.[118] provide a performance comparison between SNMP and a XML based management systems. The measurements were performed using a plain XML and a zlib compressed version of XML and showed a considerable overhead for XML processing. At the time the paper was written, binary XML implementations like EXI were not available. As our results in Section 3.2 show, EXI provides superior performance compared to an approach based on compressed XML. This characteristic of EXI is an important prerequisite for the implementation of XML based techniques on resource constrained devices.

To perform conversions between SNMP and XML, our work leverages the work done in the area of SNMP to XML mapping. We use a model-level mapping to translate between the SMI and the XML data model. Our implementation shows that it is possible to combine a SNMP based management interface and a XML based management interface in a single solution that can be implemented efficiently on resource constrained nodes. This is possible due to a carefully designed information model and query language that is backwards compatible to SNMP.

Management Protocols for Embedded Networks Lim et al.[92] propose a SNMP proxy for wireless sensor networks. Opposed to our solution, the SNMP functionality is implemented inside the proxy and not on the individual nodes. The nodes submit data to the proxy which stores the information in a log file. The proxy provides a SNMP based interface to this log file. This approach allows only read access to sensor devices because there is no possibility to send updates to a mote. The second drawback is that the proxy is application specific. If a new node with a new sensor type is added to the network, the proxy proposed by Lim et al. has to be adapted manually to include the new sensor type. This introduces a considerable management overhead and prohibits any access to the new device until the proxy is changed. The solution we propose does not have these limitations. New devices are immediately accessible and we provide full access to the management information including the possibility to change data and invoke management methods on the remote device. Our solution requires a mediator between the IP based external network and the embedded network, too. But opposed to the proxy from Lim et al., the bridge used in our system is generic, i.e., does not have to be adapted to the available devices.

The Bridge of the SensorS (BOSS)[143] proposes a UPnP based gateway for sensor networks. The gateway mediates between UPnP, which is used as external interface to the network, and an internal protocol used for the communication with the devices in the embedded network. Like the SNMP proxy, the BOSS gateway uses a proprietary protocol for the communication with the nodes in the embedded network. This is problematic if devices from different vendors have to be integrated into a single network. To ensure interoperability, the protocol used for the communication between the nodes in the embedded network and the BOSS gateway has to be standardized. Because this protocol has to provide the same functionality as UPnP, this standardization will ultimately lead to the definition of another management protocol. We avoid the definition of another protocol (tailored for embedded networks) by adopting SNMP to the resource restrictions imposed by embedded devices.

A lot of work has been done in the area of network management for monitoring oriented wireless sensor networks (WSNs). Networks that follow the "Smart Dust" vision, i.e., are composed of a large number of nodes with high redundancy, require different management functionality compared to the control oriented networks we are considering in this work. Lee et al. [87] provide an overview of projects in this area. Typical tasks of the management layer in WSNs is to collect information about the battery status of nodes, the network topology, link characteristics such as signal strength and bandwidth, and general characteristics such as the coverage of the observed area. Based on this information, typical management tasks are the (re-)configuration of the communication in order to compensate node failures, the adoption of sampling frequencies, the power management of nodes, etc.

The motivation for the development of new management protocols for embedded networks was the observation that WSNs require a new management philosophy. The user should be provided with a management interface on the network level instead of the node level. In this vision, a user only specifies the intended high level changes; the actual reconfiguration required on individual nodes is handled transparently by the system. This management interface is often tightly integrated with the application. This is possible because nodes in a monitoring oriented network are based on an identical hardware and are all executing the same application code. The required management functionality and the used hardware are known during system development and can be integrated directly into the application.

A project that falls into this category is the Management Architecture for Wireless Sensor Networks (MANNA)[125]. It provides information about the network topology, energy resources, sensor coverage and other characteristics based on a specialized management protocol called MANNA network management protocol (MNMP)[124]. The authors also provide some functionality to handle faults in event driven networks[126].

The Sensor Network Management System (SNMS)[154] provides health monitoring for WSNs. It offers a query interface that allows an user to retrieve health information from the network and a logging system to store events. SNMS comes with own protocols for data collection and dissemination to ensure an independence of the managed application. SNMS realizes network level management by supporting a dissemination algorithm that allows a delivery of control messages to multiple nodes at once. This differentiates SNMS from point-to-point management protocols such as SNMP.

The Sensor Network Management Protocol (sNMP)[23] offers management functions for retrieving the network topology, energy map and usage patterns. For collecting this information, the authors propose a topology extraction algorithm called STREAM[24]. sNMP is focused on data collection. The authors state that the constructed routing tree for data collection can also be used for data dissemination but provide no further details on this issue.

Louis Lee et al.[89] propose a policy based management system called Wireless Sensor Network Management System (WinMS). It supports an automatic adaption of the network based on trigger conditions. If for example measurements are above a certain threshold the WinMS system can automatically assign bigger transmission slots in a TDMA protocol (FlexiMAC[88]) to allow a transmission of measurements with higher data rates.

Besides these projects there are other projects that provide specialized functionality for debugging, power management, traffic management, etc. The solutions are tailored for the specific application field, whereas we aim at a general management solution. We will therefore not present these approaches in detail. A general overview can be found in Lee et al.[87].

The prerequisites that fostered the development of these new management solutions are not given in control oriented networks. Control oriented networks are heterogeneous. Each node may possess different hardware devices and execute different services. The required management functionality is therefore not known beforehand and may even change during the lifetime of a network, when new devices and services are added. Due to the diversity of nodes and the simultaneous execution of multiple control tasks, a network wide management interface is not feasible for control oriented networks. Their requirements are much closer to the management requirements known from IT networks and protocols known from this domain can be leveraged.

Note that the ϵSOA platform offers network wide management interfaces through the model driven development approach. Changes in the model are automatically and transparently mapped to reconfiguration options in the embedded network. In order to communicate these changes to the individual nodes, the management protocol presented in this section is used.

Cross-Layer Information Sharing In the recent years, several approaches for cross-layer communication stacks have been developed, especially in the area of embedded networks. An overview covering many of these approaches is presented by Razzaque et al.[122] and Srivastava et al.[145]. The possible cross-layer solutions span a wide area. One research field are solutions that propose architectural changes, such as combining several layers of the network stack or partitioning the network stack based on other criteria. Other solutions propose new interaction possibilities between the layers of a network stack. The last group are information sharing approaches. These approaches typically do not change the layered structure of the network stack. Instead they provide an interface that allows the different layers to access a shared information base. We will focus on the research projects directed at this last area, as these provide functionality comparable to the cross-layer communication interface in the ϵSOA platform.

The MobileMan project [21] proposes a networks stack that contains a shared information database called network status (NeSt). This shared storage is made available to all layers of the network stack through a publish/subscribe API.

CrossTalk[168] provides two views of the network protocol stack. A local view that contains information available at the local node and a global view that provides network wide information. The latter is created by using a data dissemination protocol that distributes local information throughout the network.

An architecture similar to CrossTalk is used in XLENGINE[7]. It also offers an interface to local information and a module for the dissemination of information into the embedded network in order to create a global view w.r.t. specific metrics.

The TinyCubus project[101] proposes a cross-layer framework for sensor networks. Cross-layer information is stored in a State Repository and can be shared between different layers of the network stack. The type of the shared information can be specified with a specification language to allow a customization based on application requirements.

In X-lisa[103], information sharing between multiple layers of the network stack is implemented using three tables containing information about the neighborhood of the node, the data sinks consuming data from the node and the messages exchanged by the node. Components can access these tables through an API that allows the retrieval and update of individual entries.

The primary motivation for the development of the cross-layer communication interface in the ϵSOA platform was to support an information exchange between the application layer and the network stack. An important requirement in this setting was the possibility to dynamically change the type and structure of the shared information. Applications may be installed and removed at any time and the cross-layer information repository has to be flexible enough to reflect these changes. This distinguishes our solution from the approaches above, which all assume a static set of shared information. The only exception is the TinyCubus, which also allows to changing the type of shared information. Our prototypical implementation shows that a XML based information model and a XPath oriented query language can be implemented efficiently on resource constrained devices. Despite these differences, there are a lot of possible synergies between the ϵSOA cross-layer interface and the related projects. The ϵSOA platform currently uses a pull based mechanism to retrieve the network topology from the embedded network. The functionality provided by the global view in CrossTalk and similar protocols is a promising extension for a push based solution that provides faster reaction times to changes in the topology. In the other direction, these projects could benefit from the extensible information model used in the ϵSOA platform.

Topology Discovery The focus of our work was the design of a generic cross-layer communication interface to allow an easy integration of the different optimization techniques analyzed in the projects mentioned above. To showcase the functionality we implemented a basic topology discovery mechanism. It performs reasonably well for the test environments used in our prototypical implementations. But there are further optimization potentials. Topology discovery has been studied for quite some time and there are many - sometimes application specific - solutions. Topology discovery for IP networks has been studied for several years now, a comprehensive survey is provided by [33]. We will only present an overview of projects targeting wireless embedded networks.

A task closely related to topology discovery is sensor localization. Localization algorithms are used to determine the absolute or relative position of nodes in the environment. If localization information is available, it can be used to create an initial network topology based on communication ranges, which can be refined at runtime to incorporate obstacles that influence the communication between nodes.

Farrell and Davis[39] propose a topology discovery algorithm for camera networks. The authors track the movement of mobile objects and derive neighborhood relationships based on the point in time when each camera detects an object. If a slow moving object leaves the observed area of camera A and is detected by camera B a few seconds later, one can assume cameras A and B are neighbors. These observations can not only be used for creating a topology of the network, but also to provide estimates about the movement of objects.

A comparable approach is pursued by Marinakis et al.[98]. They divide the topology discovery into two phases. In the first phase, the association between sensor observations and motion sources is inferred. In the second phase, network connectivity parameters are determined in a way that best describes observed transitions between nodes.

A multi-modal localization algorithm for wireless sensor networks is proposed by Farrell et al. [40]. With the help of two cameras, the position of wireless nodes in an observed area is calculated. This position can be used to derive the network topology and calibrate the sensor devices. When the calibration is finished, the nodes can be used to detect and track the movement of objects through the observed area.

Maróti et al.[99] present a localization system called Radio Interferometric Positioning System (RIPS). The authors use pairs of nodes that simultaneously emit radio signals with different frequencies. Based on the phase difference between the received signals and the signal strength, a node can determine its position relative to the sending nodes. The benefit of this approach is that the radio module is sufficient and no additional hardware is required on the receiving nodes.

A distributed algorithm for node localization in sensor networks is described by Moore et al.[104]. The algorithm is capable of determining the location of nodes based on noisy range measurements. The computation is fast enough to allow a localization of mobile nodes.

An active research area in embedded networks is topology control. Topology control algorithms are used to control the communication of nodes in a wireless network in order to guarantee some graph property (e.g. connectivity) and simultaneously reduce the overall power consumption and/or interference between nodes. Santi[128] provides a good overview of projects on this topic.

If required, the topology discovery in the ϵSOA platform can be extended with or replaced by these solutions. Due to the cross-layer interface, only a minimal amount of changes are required. These are limited to the optimization algorithms using the network topology, which have to incorporate the new/changed topology metrics. On the nodes, the addition or removal of topology metrics can be done dynamically in the information model and requires no implementation changes.

Summary and Possible Extensions The communication efficiency in embedded networks can often be improved by relaxing the strictly layered design of network protocol stacks. A communication interface that allows a communication between different layers can provide each layer with additional information that can be used to optimize and adapt the communication based on application requirements and network characteristics. Because the exchanged information depends on the application field and the involved components on each node, the interface has to be adaptable. At the same time, the overhead for retrieving or publishing data should be low to allow an implementation on resource constrained nodes.

Another functionality that is crucial for an efficient management of mid-size and large scale installations is a management interface that allows an easy inspection and adoption of the configuration of individual nodes in the embedded network. In the IT domain, the Simple Network Management Protocol (SNMP) is widespread used for this purpose.

A closer look at SNMP reveals that the required functionality is closely related to the functionality required by the cross-layer communication interface. In this section we proposed an information model based on a tree data structure that supports efficient cross-layer data exchange and can be used to implement network management functionality. Nodes in the information model can be selected with an XPath oriented expression language. A selected node can be queried to retrieve the stored information or updated to change the stored information. The information model contains an overview of all information in the embedded network. A node has access to both, information published by other components installed at the local node and information published by components on remote nodes. The access to local information is very efficient and can be used to implement a shared information repository between the different layers of the network stack. The access to remote information is handled transparently by the system and can be used to collect network wide information, such as the topology, with an easy to use interface. The second use case for remote access is network management. Based on the information model, configuration parameters and status variables of any node in the embedded network can be retrieved and modified. Nodes may additionally offer methods. These methods can be invoked using the ePath language to perform more complex management tasks.

The proposed solution can be implemented with a very low resource consumption and is feasible even for resource constrained nodes. Through a bridge that performs a message conversion, a SNMP compatible interface and a Web service based management interface can be provided. The main purpose of the bridge is the transformation between different network and transport protocols, because embedded networks often do not support the IP protocol. To get a lightweight implementation of the bridge, the information model was carefully designed w.r.t. two goals. It is backwards compatible to SNMP and uses XML technologies.

The compatibility with SNMP is achieved by using a path expression language that is semantically identical to the identifiers used in SNMP. The resulting binary path expressions are valid SNMP object identifier and vice versa. The only exception are extensions that were introduced to overcome some limitations of SNMP: the possibility to invoke methods on objects and the possibility to store structured data in tables. These features are not supported directly by SNMP and are nowadays implemented by using workarounds. In the current implementation the user has to implement these workarounds manually if a SNMP based interface is required. Otherwise the corresponding subtree of the information model is only accessible via the Web service interface. A possible extension would be to integrate the generation of these workarounds in the bridge.

The Web service bridge offers a REST interface for network management. The information model was designed to support such an interface out of the box. The ePath expressions can be supplied directly via a URL in the request. The returned results are valid XML documents (expressed in EXI) and can be returned to the requester without further processing. The Web service bridge is therefore very lightweight. Its sole purpose is to convert between http requests and the ϵSOA message format. If the requestor does not understand EXI, the bridge can transform the result document to a plain XML representation.

Besides the points already mentioned, there are some other extension possibilities. The implementation currently only supports a pulling of information. If a request is issued periodically, a push based approach might be more efficient. A possible extension to the presented solution is the addition of a subscription mechanism, comparable to the concept of traps in SNMP. If an update is received for a node, every subscriber of this node is informed.

The cross-layer interface provides the technical foundation for exchanging information between the application layer and the network stack. The ϵSOA platform implements two optimizations based on this information: topology-aware execution of applications and self-adjusting live signals. In ongoing work, we are investigating further optimizations. An interesting research area is the addition of Quality of Service metrics, such as latency, to the System Model. In the application layer, this information can be used to calculate an estimated execution time for applications - including required network transmissions. In the network stack, QoS requirements supplied by the application layer can be used to automatically configure time driven MAC protocols or dynamically prioritize important or urgent messages.

3.9 Dynamic Installation of Services

The flexibility of embedded networks can be increased, if new services can be installed at runtime on the nodes in the embedded network. If a new application should be executed by the embedded network, or an improved version of the used services becomes available, the new software can be distributed on the nodes with only a small downtime for the reconfiguration of the network. This feature is especially important if devices in the embedded network are not easily accessible, such as devices installed at a remote location.

3.9.1 Dynamic Service Installation in the ϵSOA Platform

A prerequisite for a dynamic installation of services is that the operating system running on the nodes supports the dynamic loading of code. Not all operating systems for microcontrollers possess this feature. If this is the case, the only way to install a new service on a node is to reprogram the whole node. In most cases, the reprogramming requires a reboot of the node and the internal state of all service instances is lost during this process. The ϵSOA platform possesses some features that ease the update of nodes in such a situation. Using the migration interfaces, it is possible to extract the persistent state of services running on a node. This state can be saved on other nodes in the network during the reprogramming and can be used to restore the state of the node after the reprogramming has been finished. Another feature offered by the ϵSOA platform are service libraries. Services contained in the service library are pre-installed in the unused program memory on the nodes whenever a node is programmed. If the user wants to install a new service on a node, the service library on this node is checked. If it contains the desired service, it can be instantiated without a reprogramming of the node. This mechanism can be used to support a dynamic relocation of service instances, even if no dynamic loading of code is supported by the underlying operating system. If the operating system supports the dynamic loading of code, new services can be downloaded on the node at runtime and instantiated without stopping running applications.

Services are shipped in bundles in the ϵSOA platform. A service bundle comprises the following components:

- The service executable (which includes the generated EXI message parser)
- The service description document (the eSDL)
- The management information model of the service

These components are installed on the node using a reliable communication channel (e.g. a TCP connection) between the node and the repository containing the service. The node stores the service and the accompanying documents in its flash memory. When the download of all components has been finished, the service is registered at the node. During this registration, the service implementation is loaded from the flash to the program memory and initialized. From this point in time, it can be accessed like any other service installed at the node. To use the service in an application, a new instance of the service can be generated using the management interface provided by the node.

3.9.2 Related Work

The distribution mechanism described above is fairly simple. It was motivated by the observation that many services are installed only at few nodes in an embedded network used for control and automation tasks. In many monitoring oriented embedded networks, all nodes execute the same services. In such a scenario, the individual update of each node is too time consuming and uses too many network resources. As a consequence, special code dissemination algorithms have been developed which allow the efficient reprogramming of whole networks based on a single source containing the new service(s). An overview of research activities in this area can be found in Han et al.[50] and Wang et al.[119], we will only present selected projects in detail here.

A code update mechanism targeting all nodes in an embedded network has to deal with several complexities introduced by (wireless) embedded networks. Nodes are severely resource constrained, may not be available when the update process is started and can fail during the update process. The communication in embedded networks, especially if it is based on wireless links, is unreliable and message loss will occur. As a consequence, a retransmission of packets has to be supported by the code dissemination protocol. Several multicast code distribution schemes have been proposed that target such environments. Probably the most prominent one is Deluge[54]. Deluge is a density-aware protocol, i.e., has mechanisms to reduce the amount of exchanged control messages in regions of the network with a high node density. Furthermore, Deluge is an epidemic protocol, i.e., will quickly spread data throughout the network and has built-in mechanisms to compensate lost messages. By combining both concepts, data can be distributed reliably to large numbers of nodes with a fairly high data rate.

FlexCup[100] provides the possibility to distribute only parts of the images installed at a specific node. This can result in considerable savings w.r.t. the number of bytes transmitted over the network. Updates in the ϵSOA platform are always targeting single services, so this functionality is not needed during the installation of new services for platforms that support dynamic code loading. The FlexCup approach is beneficial for updating TinyOS based nodes or other operating systems that support no dynamic code loading. In these cases, it allows updating only the modified services and keeping the remaining part of the runtime on the node untouched. We are planning to analyze this feature in more detail in the future.

3.9.3 Ongoing Work: Peer-to-Peer Code Distribution

The code dissemination algorithms mentioned in the previous section cannot be used for the distribution of services in control oriented networks. Because only a few services will need the new service, the overhead for submitting the code to each node is too high. On the other hand an update of individual nodes from a central repository is often not optimal, too. Assume we want to move a service from one node to another. In this case, at least two nodes in the embedded network possess a copy of the service, the service repository and the source node of the migration. If a service is installed at multiple nodes, this will be the case for even more nodes. An interesting research question is, how these copies can be used to improve the performance of the code dissemination in control oriented embedded networks.

The code dissemination problem has several analogies to peer-to-peer file sharing known from the IT domain. One or more sources in the embedded network possess a copy of a resource (the service). At any point in time, a node can issue a demand for a specific resource. This demand should be supplied as efficiently as possible, thereby avoiding bottleneck situations and an overloading of single nodes. A prominent protocol used for file sharing is BitTorrent[19]. The basic idea of BitTorrent is to split a file into several chunks. These chunks are initially located at a single node, the original source of the file. If a node is interested in receiving the file, it will start to download chunks from other nodes in the network. Whenever the download of a chunk has been finished, the chunk will be immediately offered to other nodes. This guarantees that the original source will not be overloaded. Even more, it ensures that the number of nodes offering a specific chunk will scale with the interest in the corresponding file. If more nodes are interested in downloading a file, more nodes will automatically also start offering parts of this file for other nodes. The information about which node possesses which chunk is maintained by so called BitTorrent Trackers, which keep a list of nodes offering a copy of a certain chunk.

We are currently investigating how such an approach can be ported to an embedded network setting. The idea is to split a service into multiple chunks that can be fetched from different nodes in the network. This would allow distributing the overhead for code dissemination over multiple nodes. A possible extension would be that nodes with sufficient storage capacities host copies of services or some chunks of a service to further reduce the processing overhead for each individual node. The critical part for the implementation of this approach is the implementation of the tracker, which maintains the list of service chunks stored at each node. We are currently investigating how this tracker can be implemented efficiently on resource constrained devices. There is some previous work, however with another focus, that shows that an implementation of BitTorrent is possible in embedded networks[43]. Another interesting research question is how the wireless communication influences the behavior of BitTorrent like protocols. Because wireless links broadcast data to all nodes in range, multiple nodes will receive a copy of a single chunk without additional costs. Another research question we are looking at is, how this broadcast property influences the performance of BitTorrent and what benefits can be realized with a peer-to-peer code dissemination compared to the simple solution using a single update source.

3.9.4 Summary

In this section, we gave an overview of the dynamic installation of new services on nodes. Code updates in the ϵSOA platform are performed on a per node basis. This design decision was motivated by the observation that nodes in a control network typically have highly different tasks. Because of this diversity, a specific service is installed only at a small number of nodes. The small number of targets prohibits the use of epidemic dissemination protocols often used in monitoring oriented networks. In this situation, a bunch of single node updates will provide a better overall performance.

We also outlined a possible solution using a BitTorrent like code dissemination protocol, which we are studying in ongoing work. This protocol could fill the gap between protocols that aim at updating all nodes in a network (e.g. Deluge) and single node updates and would be a perfect match for the code update requirements imposed by control oriented embedded networks.

CHAPTER 4

Advanced Features

Based on the basic functionality provided by the ϵSOA runtime, we implemented some advanced features that provide additional functionality for application developers. The first extension are failure compensation mechanisms, which are shown in Section 4.1. All mechanisms can be executed in-network and can be used to achieve a continuous operation until the failed components have been replaced.

If the underlying network structure changes (or if a node fails), the initial placement of services can become sub-optimal. In these cases, an optimal execution of applications can often be achieved by relocating service instances between nodes. In Section 4.2 we will present service migration strategies that allow the relocation of stateful and stateless service instances at runtime.

The integration of embedded networks and IT systems becomes increasingly important. This requires not only a conversion of different message formats and communication protocols, but also a mediation between different service execution and composition paradigms. In Section 4.3 introduce the ϵSOA service bridge, which provides this functionality.

4.1 Failure Compensation

In embedded networks, the handling of node and link failure is greatly influenced by the probability of their occurrence. In the envisioned "smart dust" networks, node failures are a common phenomenon. The reasons for such failures are manifold: energy depletion, environmental influences, mobile nodes that leave the communication range of the network, etc. The network as a whole is only functional due to the massive redundancy provided by the large number of available nodes. In such a scenario, an explicit reconfiguration of the embedded network upon each node failure is not feasible. Instead, failure compensation mechanisms are integrated directly into the used network protocols and the applications executed in the embedded network. Examples are data-centric routing (e.g. Directed Diffusion[64]) and data processing mechanisms (e.g. Geographic Hash Tables[136]). A data-centric approach is focused on the data itself, which device/node acquired the data is not important. By storing data redundantly on multiple nodes, a node failure is much easier to compensate compared to node/sensor centric processing schemes, which route and process data based on the device that acquired the data.

Control oriented embedded networks offer a much higher stability. In many cases, a redundant installation of actuator devices is not possible or too costly. Furthermore, nodes in control oriented networks are often specialized for a specific task. Simply adding some additional general purpose nodes will not provide the required redundancy to handle arbitrary node failures. As a consequence, nodes in control oriented networks are based on much more reliable nodes compared to the above mentioned “smart dust” networks. Nevertheless, node and link failures can still occur and have to be compensated by the embedded network to ensure a continuous execution of applications.

There are many failure scenarios with different levels of complexity. A basic assumption that guided the development of the failure compensation mechanisms presented in this section was that the failure compensation should provide an intermediate solution that ensures a continuous operation of applications until the failure is remedied by a replacement of the failed components. We therefore aimed at designing a failure compensation mechanism that is capable of dealing with single node failures. If the failure of multiple nodes or complex scenarios such as network segmentations should be supported, the mechanisms presented in this section have to be extended. The second goal was to design a resource efficient failure compensation mechanism. Embedded networks comprising solely resource constrained nodes should be capable to autonomously recover from a failure. The presence of a powerful node should not be a prerequisite for the proposed solution.

We will first take a closer look at link failures and how these are handled in the ϵSOA platform. After that we will present the node failure compensation in more detail.

4.1.1 Link Failures

In the ϵSOA platform, link failures are handled in a two phase procedure. If a link fails, the ϵSOA platform initially relies on the used network protocol to compensate the failure by choosing another route in the network. If the failure persists for an extended period of time, the optimizer component in the ϵSOA platform is informed about this situation. This is done by updating the network topology through the Cross-Layer-Interface presented in Section 3.8. Based on the new topology, the Optimizer can re-optimize the execution of applications, for example by moving services from one node to another. The separation of the link failure handling into two phases has several benefits. The network protocol can react autonomously and immediately on broken links. This ensures that a fast reaction on link failures is possible. This initial compensation by the network protocol is invisible to the applications executed in the network. If the link failure is transient, e.g. a disturbance caused by an object moving through a wireless link, it can be handled completely transparently by the network protocol. A reconfiguration is only triggered if the network structure changed permanently. The global reoptimization performed by the optimizer ensures that the execution of applications is always optimized for the current network structure, even if the network undergoes massive reconfigurations after the initial installation.

Note that the same mechanisms used for handling link failures can be used to integrate new nodes (with new communication links) into an existing embedded network. New links are first detected and used by the network protocol and trigger global reorganizations in the Optimizer if they persist.

4.1.2 Node and Instance Failures

Node failures can be compensated in the ϵSOA platform using three mechanisms:

- Implicit compensation by the application
- Graceful degradation
- Redundancy

Some control tasks implicitly support the compensation of failed nodes. An example is a PID-control loop. If multiple actuators are attached to the controller, a failed device will be compensated by a higher output of the remaining actuators. The ϵSOA platform offers a notification interface that allows services to inform the user about an unexpected system behavior, e.g., when the PID-controller detects that it has to use an unexpectedly high output value[1]. The same interface can be used to report malfunctioning sensor devices, e.g., sensors that show an erratic behavior or a high deviation from other sensors observing the same phenomenon.

A graceful degradation can be used if the failed component is not essential and the affected application can continue to work - maybe with reduced performance or functionality. Consider a smart lighting application that dims lights according to the measurements of a brightness sensor. If the sensor fails, the lighting application can continue to work. However, the functionality is reduced: the lights will be either turned on or off, an automatic adjustment based on the brightness is not possible anymore. The user can annotate services during application development to specify non-essential services for each application. If a failure of such a service is detected, the application is notified and can change its behavior. Analogously, the application is informed when the failure is remedied. When an essential service fails, the application will be stopped - unless a redundant replacement is available.

Redundant replacements are the third compensation strategy supported by the ϵSOA platform. During application development, the user can specify groups of

[1] In the Smart Lighting Demonstrator shown in Section 6.2 this feature can be used to detect aged or failed lights, which are less bright or not working at all.

redundant service instances. This information is stored in the service choreography description of each application. If an instance/node fails, the network autonomously replaces the failed instance with a redundant instance and continues to execute the affected applications.

The latter two compensation mechanisms are only possible, if failed nodes can be detected. We will first present a mechanism for detecting such node failures. After that, we will present the redundancy based failure compensation mechanism in more detail.

4.1.2.1 Node Failure Detection

Node failures in the ϵSOA platform are reported through the Cross-Layer Interface described in Section 3.8. Which mechanism is used to perform the actual detection is dependent on the used network protocol. Some protocols such as IPv6 (and therefore also 6LoWPAN, which is based on IPv6) offer built-in support for neighbor detection[59]. This information can be queried from the network stack and published through the Cross-Layer Interface. The ϵSOA platform also supports a general neighbor detection/failure detection mechanism, which can be used if no neighborhood information can be retrieved from the underlying network protocol. The mechanism is based on periodic keepalive signals, which are broadcasted by each node. Based on these broadcasts, neighboring nodes can detect each other and - if the broadcast is not received - detect node and link failures. The broadcast is limited to one hop to avoid unnecessary forwarding. If no keepalive signals are received, the node is treated as failed. If wireless communication is used, the keepalive broadcasts can be omitted if the node sends regular network messages, e.g., a periodic measurement signal. These regular broadcasts can be monitored by neighboring nodes and can be used to detect whether the node is still available or not. Of course this is only possible if the periodic transmissions are frequent enough. This optimization is performed during the network configuration by the Optimizer component in the ϵSOA platform.

If a neighboring node detects a node failure, it is reported to a monitoring node. The monitoring node can then issue compensatory actions or can deactivate failed applications. Each node in the ϵSOA platform can be used as monitoring node. This provides a high flexibility and allows customizing the node failure detection based on the network characteristics and application requirements. We currently support the following scenarios:

Central Monitoring If the network contains a highly reliable node, e.g., a gateway node to a wide area network, this node can be used as central monitor node. In this case, all node failures are reported to the central monitoring node. This scenario is well suited if the distance (in number of hops) between all nodes and the monitoring node is low. A typical example is a star topology with the monitoring node as the center. Such networks are often encountered when wireless networks are attached to a wired backbone, e.g. rooms in a smart building. The gateway node, which connects both networks, is a good candidate for a monitoring node. It will typically have a reliable power supply and will be located in close proximity to all nodes.

Distributed Monitoring A drawback of the central monitoring approach is the central component, which may turn into a single point of failure in some situations. The distributed monitoring remedies this situation by defining different monitoring nodes for each application. If possible, the ϵSOA platform will select a monitoring node that executes no service from the monitored application. This ensures that single node failures will either be detected by the monitoring node (when a node used by the application fails) or do not disturb the execution of the application (when any other node fails).

4.1.2.2 Node Failure Compensation

A monitoring node has access to the service choreography documents for all monitored applications. If a node failure is reported, the monitoring node first checks which applications are influenced by the failure and creates a list of failed service instances for each application. The monitoring node tries to compensate the node failure through redundantly available services. It checks all port groups to determine whether a replacement for a failed instance is available. If that is the case, the monitoring node will reconfigure the Stream Routers to send data to the redundant service instance instead of the failed instance. If a replacement of the failed instance(s) is not possible, the monitoring node will check whether the instance was critical for the execution of an application or not. This is done by analyzing the Stream Groups specified in the service choreography document. If the application is still functional, the application is informed about the failed component. If a critical service failed, i.e., at least one Stream Group is non-functional, the whole application is shut down. This is done by stopping the execution of all instances involved in the application.

4.1.2.3 Node Replacement

If a failed node becomes available again or is replaced by a new node with identical configuration, a mechanism similar to the node failure mechanism is triggered. After a new/repaired node has been discovered by the embedded network, a corresponding notification is sent to the monitoring node. The monitoring node performs the same steps as mentioned in the failure case; the only difference is that it will now try to reactivate failed applications.

4.1.3 Related Work

There is not much related work concerning failure management for embedded networks besides the projects mentioned in the beginning of this section. Gummadi et al.[49] propose a failure recovery mechanism based on a checkpointing systems that allows dealing with faults that occur during the execution of an application. The approach presented in this work tackles faults at a different level. Gummadi et al. describe a checkpointing mechanism that can be used by the application developer to design fault tolerant applications. In contrast to this, our solution is transparent for the application developer. The checkpointing approach can guarantee that a computation creates the desired result - even in the case of faults - however this comes with a considerable cost for establishing the checkpoints during the execution of the application. Our approach introduces no overhead at runtime, but the execution of applications will be disturbed for a short time until the reconfiguration of the network has finished.

Balfanz et al.[6] describe a tracing mechanism for monitoring oriented networks that continuously reports measurements to a sink node. The tracing mechanism allows learning the network topology based on neighborhood information that is piggybacked to periodically submitted measurement data. Our work assumes that the topology of a control oriented network is known during the configuration of the network. This information is exploited to optimize the execution of applications and also to optimize the fault handling by selecting appropriate monitoring nodes for each application.

4.1.4 Summary

In this section we presented the failure detection and compensation mechanisms supported in the ϵSOA platform. The first alternative is the implicit failure compensation by the application logic. The ϵSOA platform offers a notification interface that allows applications to report errors and inform the user about unexpected system behavior. The second alternative is a graceful degradation of service/application functionality. When a non-critical component of a service composition fails, the remaining services are informed and can adapt their behavior. The third alternative is a replacement of failed service instances through redundantly available instances. Redundancy information can be specified during application development and is stored in the service choreography. Based on this information, a monitoring node can autonomously replace failed instances.

4.2 Service Migration

The infrastructure of embedded networks is not fixed. New nodes can be added, existing nodes can fail and link characteristics can change. To ensure an optimal execution of applications, the ϵSOA platform continuously monitors the underlying network. If changes are detected, the placement of instances can be optimized to adapt the execution of applications to the changed network structure. A typical operation during the optimization process is the migration of service instances, i.e., the relocation of instances from one node to another. Two boundary conditions have to be ensured by the migration algorithm: the state of the moved instance must be preserved and the migration has to be performed transparently for the application.

The migration of instances in an embedded network is closely related to the migration of virtual machines (VMs) in the IT domain. We will first present an overview of migration techniques for VMs and discuss their applicability in the context of embedded networks. We will then present a migration algorithm for embedded networks and an extension for supporting software upgrades in embedded networks. We conclude the section with a summary, related work and an overview of ongoing work.

4.2.1 Migration Techniques in the IT Domain

Different migration techniques have been studied for hardware virtualization systems. The decoupling of software and the underlying hardware offers a new degree of flexibility in hosting environments. Whole operating systems can be moved between hosts, including all running applications and configuration settings. By moving VMs, overload situations can be resolved at peak times. In low-load situations, operating costs can be reduced by consolidating virtual machines on a reduced number of hosts and shutting down idle systems. In order to make the migration process transparent to the end-user and the applications executed inside the virtual machine, established communication sessions have to be preserved and the downtime of virtual machines has to be minimized. Such a non-interruptive migration is also called live migration.

A difficulty encountered during a live migration of virtual machines is the low bandwidth of nowadays networks compared to the large amounts of main memory installed in modern computer systems. The transmission of 4 gigabytes of RAM across a gigabit network takes more than 30 seconds, even assuming no protocol overhead and an achievable utilization of 100%. In a practical setting the required time will be much higher because links cannot be used exclusively for the migration process. The simple approach to stop the virtual machine, copy its state to the new host and resume the machine after the copy is not feasible given these boundary conditions. Instead, the migration process is split into three phases:

- **Push phase:** During the push phase the original virtual machine is running while certain main memory pages are already being copied to the new host.

- **Copy phase:** In the copy phase, the original virtual machine is stopped. After that memory pages can be copied to the new virtual machine, which is started once the copying has finished.
- **Pull phase:** In the pull phase "missing" memory pages, i.e., memory pages not copied yet, are fetched on demand from the old virtual machine.

Based on these three phases different migration schemes are possible, which use one or more of the phases.

As mentioned above, a prerequisite for a transparent migration is that existing communication sessions are not interrupted. Many solutions in the IT domain use virtual IP addresses, which move along with the VM from host to host, such as Clark et al.[18] or Nelson et al.[106]. After the Copy Phase, the IP network is reconfigured to route IP packets to the new host instead of the old host. Messages that arrive during the Copy Phase can either be discarded (assuming a transport protocol such as TCP will handle the packet loss) or forwarded from the old host to the new host, e.g., using an IP tunneling technique as proposed by Bradford et al.[8]. Virtual IP addresses can be managed easily if the migration is performed inside a local network. There are also approaches that aim at extending virtual IP addresses to wide area networks[153].

These techniques cannot be applied in embedded networks. Not all network protocols support virtual addresses and many applications use non-reliable transport protocols. A migration in a heterogeneous embedded network can lead to the situation that the migrated service is accessible via a different network protocol on the new host. General forwarding mechanisms such as IP tunneling are therefore hard to apply, and often not desirable because of the resulting increase in network traffic. The other difference is that the service migration in an embedded network is performed on a different level of detail. VMs in the IT domain encapsulate whole applications; services in the embedded network are individual parts of an application. To achieve a resource efficient migration and to ensure a continuous operation of the embedded network during the migration process, the characteristics of the applications and the involved services have to be taken into account.

4.2.2 State Transfer in Embedded Networks

A prerequisite for the migration of a stateful service is that the internal state of a service is accessible. There are virtual machines that are compact enough to be executed on resource constraint devices, such as Maté[90] and the more specialized SwissQM[105]. To speed up the processing on lightweight microcontrollers Hitoshi Oi proposes a hardware based accelerator for Maté[114]. A comparable solution is used in the SunSPOT[149] platform. SunSPOTs are based on the Squawk Virtual Machine[139] and a corresponding microcontroller that can execute Java byte code "on the bare metal". The motivation for the development of these machines was the optimization of the in-field reprogramming of devices. A base image that contains the virtual machine is installed at the node and can be extended with new code that can be downloaded at runtime. The second motivation was the observation that binary image tend to be large compared to the available network bandwidth in embedded networks. A virtual machine with an optimized instruction set for the application field allows creating more compact implementations that require fewer resources for code updates. If a virtual machine is used in the embedded network, the state of services can be copied by simply transferring the state of the virtual machine.

There are also arguments that speak against a deployment of virtual machines on embedded devices. The services implemented with a tailored instruction set can be smaller compared to native implementations. However, the overall resource consumption of a VM based solution can be higher than the native alternative, because the implementation of the VM can use a considerable amount of storage on the node. It is questionable whether this amount of storage will be available in every situation. The current implementation of the ϵSOA platform does not use a virtual machine due to these reasons.

If no virtual machine is available, the internal state of a service has to be retrieved programmatically. In the ϵSOA platform, each service is assigned a memory area for storing persistent variables, i.e., variables that keep their value between method calls. This design was introduced to support the creation of multiple instances of each service. These instances share the same service implementation, but store their internal state in a private memory area. This memory area can be used to perform a state transfer between different instances of the same service by simply copying the memory contents from one node to another. The ϵSOA platform ensures that a state transfer is only triggered between invocations of the service logic. At this point in time, all state information belonging to an instance is stored in the designated memory area and can be copied to the destination.

4.2.3 Migration Algorithm

We present the different migration strategies based on the example scenario depicted in Figure 4.1. The initial setup is an application consisting of a service chain of three services (a, b and c) which are installed at nodes A, B and C. We want to move the middle instance of the service chain (b) from node B to node D. The migration is realized by installing a copy (d) of the original service instance at the new node and transferring the state of b to the copy. During this process, all data streams targeting the migrated instance (x) or originated by the migrated instance (y) have to be redirected accordingly (x' for x and y' for y).

To keep the description concise, the example scenario contains only one instance that submits data to the migrated instance (a) and one instance that consumes data from the migrated instance (c). The following migration schemes are not limited to the simple scenario and can be used to migrate arbitrary service chains. All actions concerning a and c have to be executed analogously on all involved instances.

The migration is divided into three phases, a preparation phase, the migration phase and a cleanup phase.

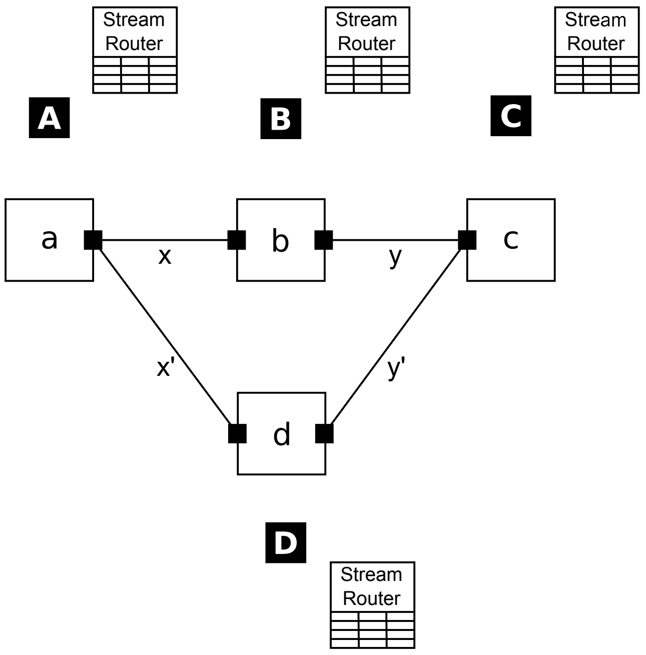


Figure 4.1: Migration Scenario

Preparation Phase The first step is to check that the target node (D) of the migration has enough resources to execute the new instance. If that is the case, the service is installed at the target node. The installation of new services is explained in Section 3.9 and not presented in detail here. The next step is to instantiate the service (resulting in the creation of d) and transfer any configuration parameters from the original instance to the new instance. Configuration information is transferred using the Management Interface presented in Section 3.8. After all these actions have been executed successfully, the Preparation Phase is finished and the Migration Phase is started.

Migration Phase The migration phase has two tasks: transferring the internal state of the service instance between the original and the new instance (b and d in the example) and adjusting the data streams accordingly. In the example the existing data streams x and y have to be removed and the new data streams x' and y' have to be installed. Data streams in the ϵSOA platform are managed by Stream Routers, which are installed at each node. Each Stream Router has a routing table that specifies the destination of incoming and outgoing data streams. Each data stream possesses an entry in the Stream Routing Table of the originating node, which specifies the destination node, and an entry in the Stream Routing Table of the destination node, which specifies the destination instance and port. To replace streams x and y with the new streams x' and y' we therefore have to remove four entries from the old streams and add four entries for the new streams.

The migration process has to be designed in a way that ensures no instance receives duplicate data. In the example, service instance c should not receive data from both b and d. This is important to ensure that events are only processed once. Assume b sends a signal to c that tells c to flip its internal state. If this message is duplicated during the migration process, i.e., sent by both b and d, c will be in an inconsistent state. Depending on the application scenario, another boundary condition is that no messages are lost during the migration process. If service a continuously produces data, e.g., a stream of periodic measurements, the loss of a few messages is tolerable. This is not the case if a produces events. Assume a is a button and the service chain is a lighting application. In this case, we want to be sure that the button press is not lost during the migration.

The ϵSOA platform offers three migration schemes. The developer can specify migration characteristics for applications and instances. Based on these characteristics, a suitable migration scheme is automatically selected. Applications can either tolerate message loss or not. Service instances have three possible characteristics. Services can be stateless, stateful or synchronizable. A synchronizable instance is a special case of a stateful instance. The state of a synchronizable instance depends on the data received during a specific time interval. A typical example are services that calculate data on sliding windows, e.g., a sliding window average. We will present migration schemes for all three service types. All schemes are designed in a way that ensures no duplicate data is created.

Scheme 1: Migration of a Stateless Instance The migration of a stateless service is the simplest migration scenario. The output of a stateless instance depends solely on the input it receives. Typical examples of stateless services are services that convert data between different formats and data types or perform filter operations. The migration workflow for a stateless instance consists of the following steps:

1. Add path x' and y' on node D.
2. Add path x' on node A.
3. Remove path y on node C
4. Add path y' on node C
5. Remove remaining entries for old paths, i.e., x and y from B and x from A

After the execution of Steps 1 and 2, both the original (b) and the new instance (d) are receiving data. Both output data streams are submitted to node C. Because stream y' has not been added to the Stream Router Table on node C, the output stream of instance d is discarded at node C. The parallel execution of b and d is not problematic because both instances are stateless. To finish the migration, the Stream Router on node C is reconfigured to discard data from stream y (Step 3) and forward data from stream y' instead (Step 4). This can be done in an atomic operation. A message loss is therefore not possible.

Scheme 2: Migration of a Synchronizable Instance The special characteristics of synchronizable instances can be exploited to simplify the migration process compared to general stateful services. Recall that the state of a synchronizable instance depends on the data received in a specific time interval Δt. After Δt has elapsed, the original and the new instance will have the same internal state if they are supplied with identical input data. This characteristic is exploited to avoid an explicit state transfer between the original and the new instance. Instead of transmitting the state, both instances are executed in parallel until their internal state is synchronized, i.e., equal. The corresponding workflow is identical to the workflow in Scheme 1 with the exception of Step 3:

1. Add path x' and y' on node D.
2. Add path x' on node A.
3. Wait for synchronization
4. Remove path y on node C
5. Add path y' on node C
6. Remove remaining entries for old paths, i.e., x and y from B and x from A

Step 3 is introduced to suspend the migration process until a synchronization between b and d has been achieved. The used delay is service specific and can be specified by the service developer. Like Scheme 1, this Scheme prevents message losses if Step 4 and 5 are executed in an atomic operation.

Scheme 3: Migration of a Stateful Instance The migration of a general stateful instance is the most complicated migration scheme. It comprises the following steps:

1. Add path x' on node D.
2. Add path y' on D and C
3. Remove path x from node A
4. Transfer state from b to d
5. Add path x' on node A
6. Remove remaining entries for old paths, i.e., x and y from B and y from C

The critical steps in this process are steps 3 to 5. It is important to execute these steps in the given order to ensure a consistent state transfer and avoid the duplication of data. After the execution of Step 3 and before the execution of Step 5, messages can be lost. In general, this period of time is comparably small because the state information of services is small and can be transmitted fast even on low-bandwidth links. If a message loss is critical, node A can be configured to store outgoing messages. The corresponding command is bundled with Step 3. After the installation of path x' in Step 5, the stored messages are delivered immediately via the new path. This ensures all messages are either transmitted via stream x and processed before the state transfer, or transmitted via stream x' and processed after the state transfer. The resulting state of d will therefore include all messages that were generated by a. This migration scheme ensures no messages are lost, however messages can be delayed if they are created at the beginning of the migration process.

Comparison of Migration Schemes What migration scheme works best for a given service is application specific. Migration Scheme 1 introduces the smallest overhead and should be applied whenever possible. There is a time window in which messages are sent to instances b and d simultaneously, what results in an increased network utilization. However this time window is small and in many cases, message duplication does not occur at all.

Migration Scheme 2 allows a loss free migration without additional memory for storing messages. It is limited to a certain class of services instances and creates some communication overhead due to the duplication of messages.

Migration Scheme 3 is the most general one, but also requires additional resources for storing messages during the migration process. Our prototypical implementation shows that Migration Scheme 3 is fast and efficient enough to be implemented even on resource constrained devices. In many cases, the internal state of services is small enough to fit into a single network packet, thus resulting in a negligible network overhead. A little more overhead is required if a loss free migration has to be ensured. Storing multiple messages can be a considerable burden for nodes in an embedded network. Our experience with the prototypical implementation shows that this problem occurs rarely in a practical setting. A loss-free message transfer is typically required for applications that are controlled by seldomly generated events. Due to the low frequency of events, the storage requirements are low. If applications use events that are generated frequently, they are often synchronizable and Migration Scheme 2 can be used.

Cleanup Phase After the migration is completed, the original service instance is deleted and any occupied resources are freed.

4.2.4 Implementation

The migration schemes are implemented with two interoperating controllers in the ϵSOA platform. Each node possesses a local Migration Controller. It triggers the instantiation and deletion of instances, handles the addition and removal of entries in the Stream Router tables and (de-)serializes instance states and transmits them via the network. The Migration Controllers are managed by a Migration Coordinator that implements the different migration schemes and issues corresponding commands to the Migration Controllers. The Migration Coordinator also stores a compensation action for each executed command. If the migration fails, these actions are used to revert the system configuration to the state before the start of the migration process. The Migration Coordinator is dynamically created for each migration process and deleted when the migration is finished.

4.2.5 Software Updates

The migration algorithm presented in this section can also be used to handle the installation of new software versions for running services. To ensure a continuous operation it is desirable to preserve the internal state of instance during a software update. A software update can be seen as a special case of a migration. The differences to Migration Scenario 3 are that the destination instance is located at the same host as the original instance. To support a state transfer between the old and the new version of a service, the ϵSOA platform offers a versioning interface. Instead of simply copying the state variables during the upgrade process, this interface is used to supply the new version of the service with the state variables of the old version. The remaining parts of the migration process are identical. Because the new instance is located at the same node as the original instance, all migration steps can be performed in an atomic operation.

4.2.6 Related Work

Pure sensor networks that perform tracking tasks often use techniques that are comparable to a service migration. We will not present these solutions here and instead focus on solutions that are tailored to the application fields and boundary conditions of control oriented embedded networks. A list of key design issues for mobile agents in WSNs is given by Chen et al.[16]. Quaritsch et al.[121] present a camera tracking application based on mobile agents that follow the tracked person through the network. The application exploits á-priori known neighborhood relationships to perform the handover, i.e., the migration of the tracking agent, between different cameras. Tseng et al.[15] describe an agent based system for tracking. In this system, objects are detected by groups of three nodes which elect a master that coordinates the tracking inside the group. If the object leaves the current group, the master moves along with the object to the next group. Because the tracking of an object is restricted to small, local groups, the communication and sensing overhead is kept low. Qi et al.[120] show that mobile agents can provide a good trade-off between energy efficient signal processing (which requires as few nodes as possible) and fault tolerant detection of objects (which requires as many nodes as possible). The trade-off is achieved by using mobile agents and local communication in the proximity of the tracked object.

From an abstract point of view, the tracking applications and the migration scheme presented in this section provide a similar functionality: both provide means to move a running service/agent between nodes in a network. However the approaches are optimized for different characteristics. Tracking applications require a frequent and fast state transfer between nodes and the relocation is typically visible for the moved agent. Services/agents used in tracking applications are often designed with this mobility in mind. Opposed to this, the migration techniques presented in this section handles migrations transparently for the involved services and applications. Any instance can be relocated without additional programming effort from the developer. In some scenarios the migration scheme presented in this section can be used for building mobile code that follows a tracked object, but a tailored solution based on mobile agents will most likely provide superior performance.

4.2.7 Summary

In this section, we analyzed the problem of moving a running service instance from one node to another. We presented a migration algorithm that allows a live migration of instances, i.e., instances and applications do not have to be stopped during the migration process. The algorithm takes into account the characteristics of embedded networks and the characteristics of the applications executed in these networks. It offers several migration schemes that can be selected based on application requirements. The algorithm was designed to require minimal intermediate storage on the involved nodes, in order to support an implementation on resource constrained devices.

4.3 Integration of Embedded Networks and Web Service Based IT-Systems

A seamless integration of services from the embedded world and services from the IT world is a key requirement for the envisioned Internet of Things and modern automation solutions. The techniques used in the ϵSOA platform were carefully designed to allow a seamless communication between both domains. But the technologies used in the communication layer are not the only difference between both domains. While SOA is a suitable architectural paradigm for both, IT infrastructures and embedded networks, the characteristics of services differ in both application domains. To mediate between both worlds, the ϵSOA platform uses a Service Bridge. The Service Bridge has two tasks. It guarantees the technical interoperability by converting between the different message formats and network protocols used in both domains. The second - and more important tasks - of the bridge is to map between the different application development and execution paradigms used in both worlds. Many of these differences have already been mentioned in previous parts of this work. We will recapitulate these differences (and the similarities) in the following section and derive a set of integration scenarios that have to be supported to allow a seamless cooperation of services from both domains. After that we will present the design and implementation of the ϵSOA Service Bridge, which offers this functionality. We conclude this section with an overview over related work and a summary.

4.3.1 Web Services and Embedded Services

The most fundamental difference between services in both domains is the data processing paradigm. Embedded networks typically use a data centric programming paradigm (often also called actor oriented or data stream paradigm). The ϵSOA platform is an example of such an architecture. Applications are composed of a chain of services that operate solely on the received data and have no knowledge about the sources of received data and the destinations of produced data. The data is exchanged between services using a push-paradigm. A crucial observation is that an application in this scenario can be realized completely decentralized.

In contrast to this, Web services are typically implemented using a request/response interaction pattern. The current state-of-the-art for composing applications from Web services is to use orchestration languages such as the Business Process Execution Language (BPEL)[112]. The BPEL process is a central coordinator that handles the Web service invocations and contains some high level processing and fault handling logic.

To distinguish both kinds of services, we will call services from the IT domain Web services and services from the embedded/automation domain ϵServices throughout this chapter.

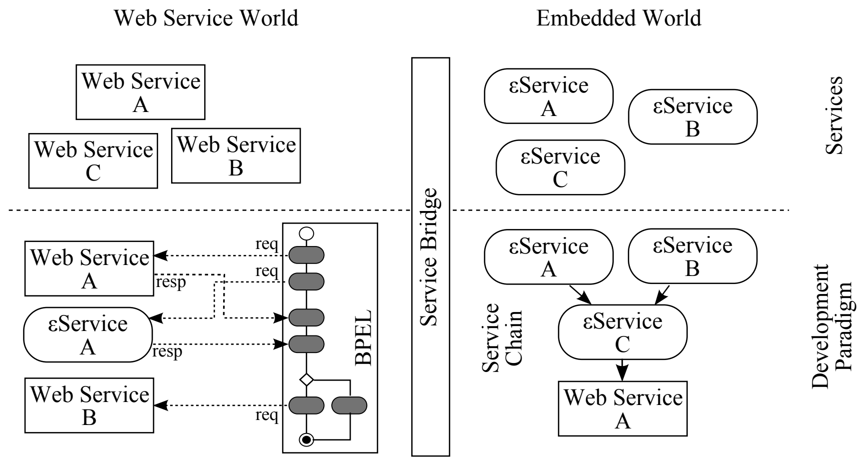


Figure 4.2: Web Services and Embedded Services - Two Views

4.3.2 Integration of Both Worlds

The challenge for application developers is the integration of both worlds, Web services on the one side and embedded Services on the other side. The integration has to be performed in two ways, as shown in Figure 4.2. A developer familiar with Web service technologies should be able to interact with services from the embedded world (ϵServices) just like he would interact with any other Web service. If, for example, a business process is modeled with BPEL [112] (as depicted in the lower left part of Figure 4.2), the process designer should be able to use ϵServices inside the BPEL process to acquire data or to submit data to field level devices. On the other hand, a developer familiar with application development for embedded networks should have access to services in the enterprise back-end in the same manner as he accesses other ϵServices. If data has to be transmitted to a back-end Web service, it should be sufficient to route a corresponding data stream to the remote service (as depicted in the lower right part of Figure 4.2).

The Service Bridge is the mediator between the two worlds: it translates messages to facilitate communication between services in both worlds and provides an abstraction layer that supports both of the above mentioned views. We will present both aspects in detail in the following sections.

4.3.3 Message Conversion

What conversions are required to allow a communication between both worlds depends on the technologies used in the embedded network. Automation system are typically not using HTTP for exchanging messages, or at least not standard HTTP but an optimized binary variant. The Web Service Bridge therefore has to translate between incoming Web service calls, which are based on HTTP, and the message format used in the embedded network. The basic architecture of the Service Bridge is shown in Figure 4.3. The main task during this translation is the mapping of ad-

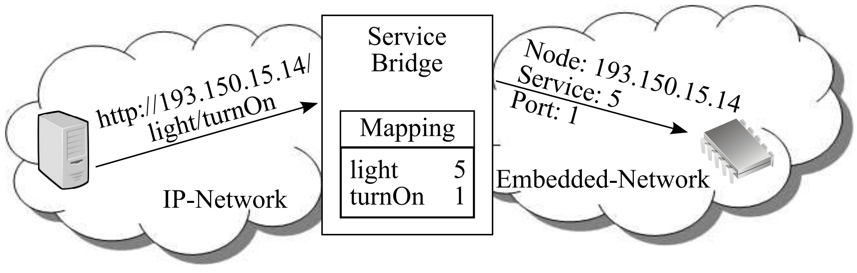


Figure 4.3: Service Bridge

dresses. URLs cannot be used in embedded networks due to their size and a numeric addressing format has to be used instead. The mapping of URLs to numeric values can be done using an indexing technique. The Service Bridge parses the WSDL documents of external services and assigns numbers to the different Web service operations based on the occurrence in the WSDL, i.e., the first WS operation is assigned the numeric address one, the second operation the numeric address two, etc. In order to make ϵServices available in the IT domain, a conversion from numeric values to URLs is needed. In the ϵSOA platform, the conversion is performed using the metadata description of ϵServices. The created URL is composed of the name of the service, appended with a number if multiple instances of the same service are executed, followed by the name of the in- or output. To get a measurement from a temperature sensor represented by service "TempSensor" a generated URL could be:

```
<address of node>/TempSensor/Temperature
```

The second aspect of the translation is the data format. The ϵSOA platform uses a binary XML representation (EXI). A design goal of EXI was to increase the communication efficiency of Web services in the IT domain. We might therefore see an increasing number of Web service stacks that offer built-in support for EXI. In these cases the data format is identical in both domains. If this is not the case, the Service Bridge will convert binary XML to plain XML and vice versa.

Note that neither of these transformations requires any service specific configuration and can be performed fully automatically by the Service Bridge.

4.3.4 Mapping of Execution Paradigms

The translations mentioned in the previous section ensure that a communication between services in both domains is possible. To map between the different execution paradigms, the following interaction scenarios have to be supported. We distinguish between ad-hoc interactions and subscription based interactions.

Ad-hoc interaction with the embedded network An ad-hoc interaction created by a Web service is a very common interaction scenario. In this case, the interaction between the services in both domains is not planned beforehand via subscriptions, but occurs dynamically. RPC-style Web service invocations are an example for this kind of interactions, e.g., in order to retrieve the current measurement value of a sensor, an external service could invoke a *getData* method on an embedded service.

Ad-hoc interaction with an external Web service This scenario is not needed in the ϵSOA platform. The stream oriented execution paradigm does not support blocking calls to remote services. In the embedded network, a request/response interaction therefore has to be modeled as a service chain. This is equivalent to a subscription based interaction with the external Web service.

Subscription based interaction with the embedded network In this scenario, an external Web service interacts with one or more a priori known services in the embedded network, e.g., to retrieve measurement values or provide externally acquired data to the embedded network. The communication is managed via subscriptions, i.e., a Web service developer subscribes to the output of an embedded service. The subscriptions are realized using existing Web service standards.

Subscription based interaction with an external Web service In this scenario, a developer from the embedded domain wants to retrieve data from or submit data to an external Web service on a repeating basis. This interaction has to support the stream based paradigm used in the embedded network, i.e., to submit data to the external service the developer routes a stream to the Web service, to receive data he routes a stream from the Web service to the embedded service.

4.3.5 Design and Implementation of the Service Bridge

The mapping of execution paradigms is implemented in the Service Bridge through *virtual services*. A virtual service represents a service from one domain in the other domain. A developer can access virtual services in exactly the same way he accesses other services in his domain. A virtual Web service (which represents an ϵService) has a Web service interface and offers Web service execution semantics, a virtual ϵService (which represents a Web service) offers a stream based interface and corresponding execution semantics. The Service Bridge offers a front-end that allows users to enable Web service based access to embedded services. If the Web service based access is enabled, the Service Bridge creates a virtual Web service and corresponding WSDL documents and reconfigures the embedded network accordingly. A similar procedure is used to enable access to external Web services from the embedded network. Based on a WSDL document of the external service, the Service Bridge creates a virtual ϵService. This service is available in the embedded network and can be integrated into service chains just like any other embedded service. The implementation of the bridge is illustrated using examples derived from the interaction scenarios shown in the previous section.

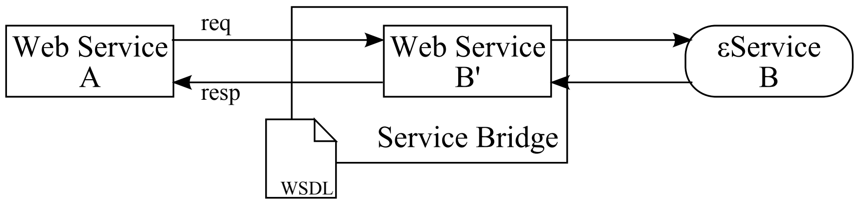


Figure 4.4: Ad-hoc Interaction with an Embedded Service

Ad-hoc Interaction with the Embedded Network A common use case for a communication between an external Web service and an ϵService is an ad-hoc request/response interaction. A Web front-end that allows users to poll sensor values or to modify actor states is a typical source of such interactions. Another example are mobile devices that are used to interact with the automation system, e.g., a PDA that is used to control the lights in a room. These interaction are infrequent and the used devices are often mobile and not always in the proximity of the embedded network. A subscription based interaction is therefore not suitable. To ensure compatibility with a variety of Web services, the Service Bridge supports three ad-hoc interaction scenarios.

Pull Based Ad-hoc Interaction

Figure 4.4 shows an ad-hoc request/response interaction scenario. In order to make an embedded service accessible from the Web service world, a WSDL generator in the Service Bridge creates a WSDL document describing the embedded service's interfaces. It contains a WSDL *Notification* type port for every output of the service and a WSDL *One-way* port for every input of the service. The correlation between the virtual ports and the ports of the embedded service are maintained in a mapping table. The newly generated WSDL is made available through a UDDI based discovery interface, which allows users from the Web service world to search for specific embedded services. The Service Bridge additionally installs data streams from the virtual service to the target ϵService.

Incoming WS requests aimed at the input of an embedded service are intercepted by the bridge and converted to messages in the ϵSOA platform. The messages are injected into the embedded network using the newly installed data streams. Any results returned by the ϵService are converted back to SOAP responses and returned to the external Web service. The pending HTTP request is stored in the Service Bridge until a response is available.

Cached Ad-hoc Interaction

The interaction scheme mentioned above requires an ϵService that allows the pulling of measurements. Sometimes such an interface is not available or the measurement is a very costly operation and should only be performed in specific intervals. In these cases, the Service Gateway offers a caching solution. If a user enables external access to a Web service, he may choose to support only cached access. If caching is enabled, the Service Bridge installs an additional caching service, as shown in

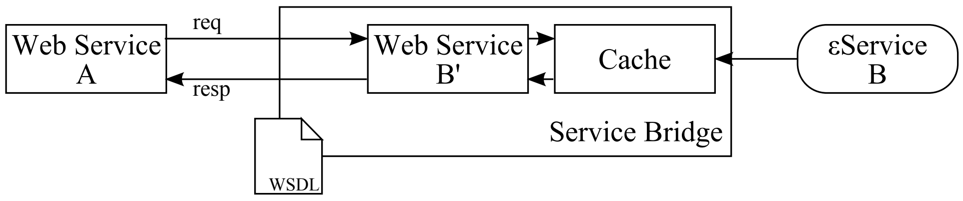


Figure 4.5: Cached Ad-hoc Interaction with an Embedded Service

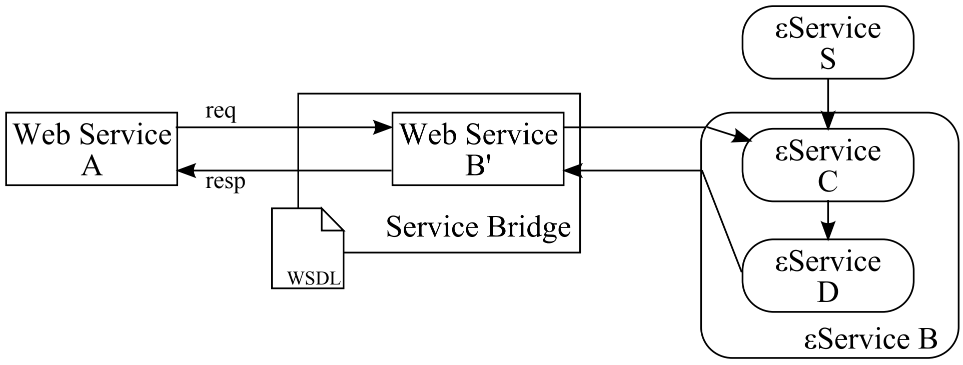


Figure 4.6: Ad-hoc Interaction with a Service Chain

Figure 4.5. The caching service has two inputs and one output. The data input is connected with the output of the ϵService. The caching service will always store the latest data received at this input. If a message is sent to the second input, the trigger input, the caching service will send the stored data. The last measurement produced by the target service is therefore pullable via a call to the trigger input. This caching mechanism can also be used to support continuous access to nodes that are not available all the time, e.g., because they activate their radio module only at specific times or are mobile and not always in communication range of the embedded network.

Interaction with Service Chains

Ad-hoc interactions are not limited to single ϵServices. The Service Bridge can also be used to provide an interface to whole service chains or parts of a service chain. Such a scenario is shown in Figure 4.6. The virtual Web service B' offers an input and an output. The input represents an input of the ϵService C, the output an output of the ϵService D. Incoming requests are forwarded to the input of C and trigger the execution of the service chain in the embedded network. The result created at the output of D is returned as Web service response to the caller. This interaction possibility is a powerful tool for building Web front-ends or Web service based interfaces for automation applications. Note that the addition of the Web service interface requires no manual changes to the application logic or the network

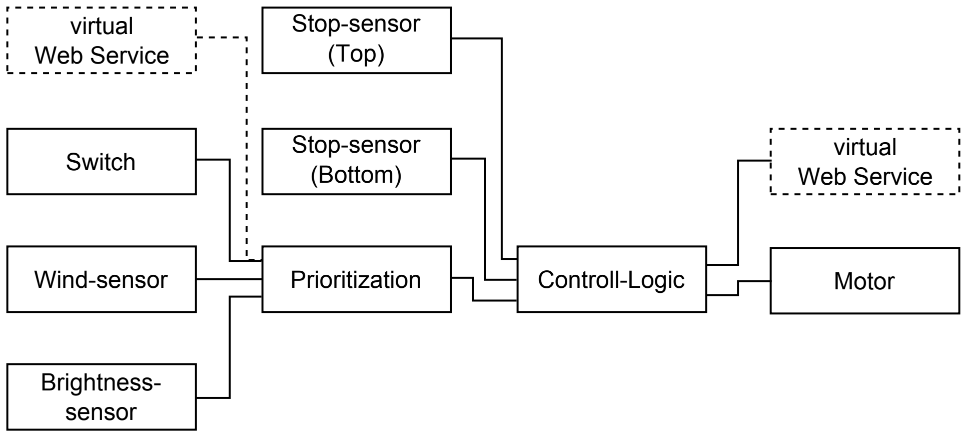


Figure 4.7:

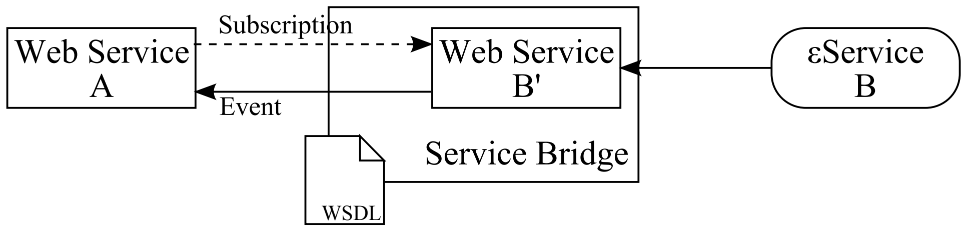


Figure 4.8: Subscription based Interaction with an Embedded Service

configuration.

Figure 4.7 shows a more complex example for an interaction with a service chain. In the example, a jalousie control application is made accessible as Web service. The virtual Web service contains an input, which can be used to move the jalousie to a specific position. This input value is submitted along with the sensor signals from other control devices to a prioritization logic. The prioritization logic ensures that manually issued commands, such as an activation of the switch or an incoming Web service call, override the commands created by the brightness sensor. The highest priority is assigned to the wind sensor, which ensures the jalousie is not damaged when high wind speeds are observed. The control application additionally uses two stop sensors to stop the motor, if the jalousie reached its top or bottom position. When the motor is stopped, the control logic reports the current position of the jalousie back to the virtual Web service.

Subscription based Interaction with an Embedded Service In some cases, an external Web service wants to monitor a measured phenomenon continuously or wants to receive continuous status updates from an embedded service. The ϵSOA

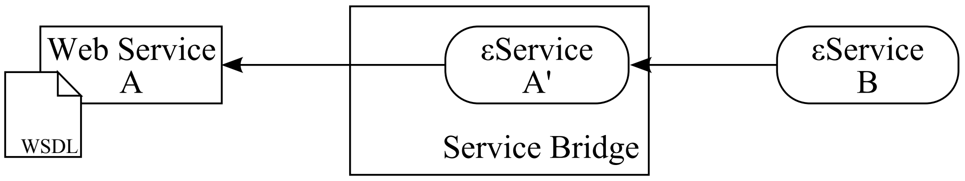


Figure 4.9: Subscription based Interaction with a Web Service

platform supports such interactions through a subscription mechanism in the Service Bridge. A corresponding scenario is shown in Figure 4.8. The subscriptions are implemented using the WS-Eventing standard. Whenever an external Web service issues a WS-Eventing subscription for an output of an embedded service, the Service Bridge installs a virtual Web service and a data stream between the output of the ϵService and the virtual service. Updates received via this data stream are converted to WS-Eventing notifications and delivered to the external service.

If a developer wants to access an external Web service from the embedded world, the Service Gateway creates a virtual embedded service representing this Web service, as shown in Figure 4.9. The virtual service's in- and outputs are created according to the WSDL description of the Web service. For continuous interaction, the *One-way* and *Notification* WSDL port types are supported. A *One-way* port in a WSDL specifies a port, which only receives messages. The virtual service will therefore possess a corresponding input. Analogously, an output is created for every *Notification* port of the WSDL. The correlation between these ports is stored in an internal mapping table in the Service Gateway. From the view of the embedded network, the virtual service is offered by the node hosting the Service Gateway. In order to send data to the external Web service, an embedded service can send data to the input of the virtual service running on the gateway node. The arriving messages are intercepted by the Service Gateway and converted to a SOAP call. The destination Web service is determined with the mapping table and the message is forwarded to its destination in the Web service domain. Incoming SOAP messages are treated analogously: they are intercepted by the Service Gateway and converted to embedded network messages. These messages are injected to the network, as if the output of the virtual service created them.

The Service Bridge can act as a WS-Eventing client and subscribe to Web services in order to retrieve external data and inject it to the embedded network. Although WS-Eventing has been standardized for quite a while, not many WS-Eventing capable Web services can be found in the Internet. Instead most Web services available nowadays offer a simple pull based API. The Service Bridge can be configured to periodically issue pull requests to Web services to allow an interaction with such services, too. The results returned by these requests are injected into the embedded network analogously to notifications created by WS-Eventing.

In many cases information from the Web is not only available through Web services but also RSS feeds or similar information sources. A possible extension of the Service Bridge is to integrate this information sources to broaden the interaction possibilities between embedded networks and Web applications.

Ad-hoc Interaction with a Web Service The Service Gateway does not directly support ad-hoc interaction with external Web services. We found no use case where this functionality is needed. It violates the data centric processing paradigm in the sensor network and many benefits of data centric systems, like free placement of services, splitting and re-using of data streams, are only achievable if the individual services operate purely data driven. An ad-hoc interaction violates this design, because it requires a decision which external Web service should be called from an ϵService. If the ad-hoc interaction is needed anyway, it can be mimicked by installing temporary data streams for the duration of the invocation. The message exchange in this case is the same as in the continuous interaction scenario.

4.3.6 Related Work

To the best of our knowledge, we are the first to propose a Service Bridge that mediates between data driven embedded services and Web services. There is a lot of literature about bridges and gateways that convert between different message formats and protocols. These approaches are tailored to the requirements of the specific protocols and - besides the basic architecture - do not have a lot of commonalities with the ϵSOA service bridge. Some techniques used in the ϵSOA service bridge, such as the mapping of URLs to numeric identifiers, are also discussed in ongoing standardization efforts concerning application protocols for embedded networks, e.g., in the IETF CoRE (formerly CoAP) working group[2].

4.3.7 Summary

In this section we presented the ϵSOA Service Bridge. The Service Bridge works as a mediator between the IT domain and the embedded domain. It converts messages between the data formats and protocols used in both domains. This ensures that a basic communication between services in both domains is possible. But this is not enough to build the envisioned Internet of Things. The IoT requires a seamless integration between services from both domains, i.e., Web services should be accessibly from applications in the embedded domain and embedded services should be accessible in Web service workflows. The main obstacle for an integration are the different service composition and execution paradigms used in both domains. Web services are connected to applications with orchestration languages such as BPEL and typically employ a request/response oriented execution paradigm. Embedded services on the other hand are organized as data centric, decentralized service chains that communicate by pushing data to subsequent services in the chain. The Service

[2]`https://datatracker.ietf.org/wg/core/`

Bridge provides a mapping between these different paradigms. Through the installation of virtual services, the Service Bridge can represent external Web services as ϵServices in the embedded domain and vice versa. These virtual services can be integrated into service chains like any other ϵService or integrated into BPEL workflows like any other Web service, respectively.

CHAPTER 5

Tool Support

The different actors involved in the creation and operation of embedded networks have to be supplied with tools that ease their work. It was not possible to develop a full-fledged set of tools with the resources available for a research project. We realized basic implementations of the tools to demonstrate the overall application development workflow in the ϵSOA platform, starting from the system model down to the executed code on the nodes in the network. There are many possibilities for extensions and additional features beyond the basic implementations described in the following sections.

5.1 Planning and Management

The heart of the ϵSOA toolset is the Planning and Management tool. An example screenshot from the current implementation is shown in Figure 5.1. The Planning and Management tool allows defining the hardware and application model used in the ϵSOA platform. The developer can specify nodes, based on hardware types that define basic characteristics such as ROM and RAM size, and the communication channels between these nodes. The hardware model (see Section 2.3.1 allows to assign properties to nodes and links. This information can be used in all tools to filter entities (nodes, links, services, instances) with specific properties.

Navigation and Filtering

The Planning and Management tool provides a graphical overview of a modeled or concrete embedded network. The user can navigate through the model and inspect and modify the individual components. The tool provides several filter mechanisms that allow an efficient modeling and administration of large scale networks. A common use case for filters in a building automation scenario is a filtering based on

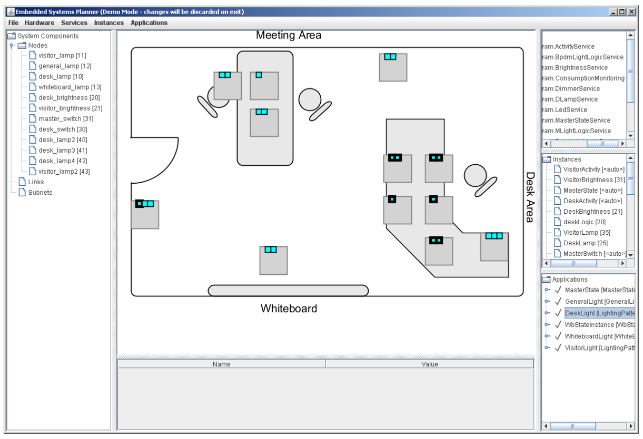


Figure 5.1: Planning & Management Tool

location information. Location information added to nodes at deployment time can be used to efficiently search for nodes in a specific room or part of the building and can be used to filter candidate nodes during the creation of service choreographies.

Definition of Services and Instances

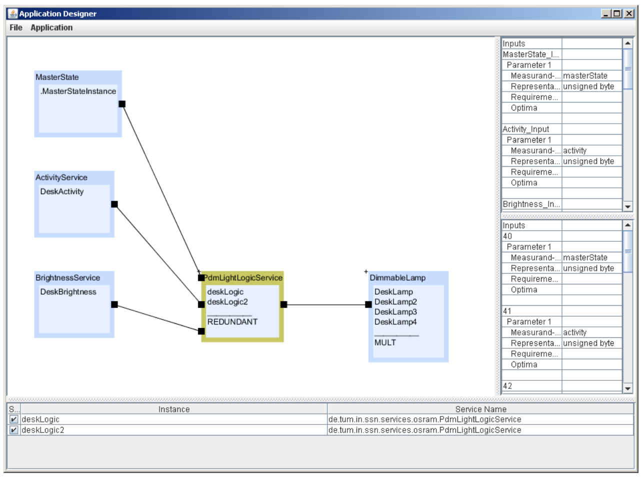
The Planning and Management tool contains a repository of services. Each entry of the repository comprises an interface definition (an eSDL document) and a service implementation. The developer can instantiate services in order to use them in applications. If the service can be parameterized, the user can configure the service instance by assigning values to the corresponding parameters.

Application Composition

The creation of applications is supported by a service composition tool that is based on a pattern based service composition as introduced in Section 2.5. The basic structure of the tool is shown in Figure 5.2. The user may load different application patterns into the tool. These are displayed in the pattern view in the center. If a user clicks on one of the slots defined in the pattern, the service view at the bottom is populated with compatible service instances, i.e., service instances that can be added to the selected slot. Like the Planning and Management tool, the service composition tool can use filters, e.g., to restrict the list of instances to instances present in a specific room. Finished service compositions are added to the Planning and Management tool.

Service Placement

Each service instances has to be placed on a node in the embedded network. The user can assign placements manually or trigger the calculation of a placement using



Inputs	
MasterState_I...	
Parameter 1	
Measurand-...	masterState
Representa...	unsigned byte
Requireme...	
Optima	
Activity_Input	
Parameter 1	
Measurand-...	activity
Representa...	unsigned byte
Requireme...	
Optima	
Brightness_In...	

Inputs	
40	
Parameter 1	
Measurand-...	masterState
Representa...	unsigned byte
Requireme...	
Optima	
41	
Parameter 1	
Measurand-...	activity
Representa...	unsigned byte
Requireme...	
Optima	
42	

S...	Instance	Service Name
✓	deskLogic	de.tum.in.ssn.services.osram.PdmLightLogicService
✓	deskLogic2	de.tum.in.ssn.services.osram.PdmLightLogicService

Figure 5.2: Service Composition Tool

the optimization algorithm described in Section 2.6. The output of the placement algorithm is a list of placements which are optimal w.r.t. a specific metric. The user can browse this list to select the most suitable one. The user may optionally alter the generated placements by manually moving service instances between nodes.

Code Generation

If all previous steps have been completed, the deployment of the modeled system can be started. The Planning and Management tools launches the model driven code generation to create images for each node in the network or to create service bundles which can be added to existing images.

(Re-)Configuration

Some changes can be deployed without a reprogramming of nodes. An example is a reconfiguration of applications, which only requires the modification of instance parameters. Another example is the relocation of services between nodes or the reconfiguration of service compositions, e.g., the replacement of a service instance through another, compatible instance. The Planning and Management tool tracks all changes made to the system model. If the user is satisfied with the new configuration, he can deploy all pending changes to the live system.

Status and Ongoing Work

The current implementation of the Planning and Management tool allows developing automation systems based on a model-first approach. The developer first specifies the target system. Based on this model, code images can be automatically generated for the nodes in the network. The Planning and Management tool can also be used to modify systems after the initial installation. Changes to the model are automatically tracked and can be deployed to the live system.

We are currently working on an integration of a node discovery mechanism into the Planning and Management tool. The ultimate goal is to store all configuration and management information on the nodes in the network. An administrator can download and access all information required for the management of an embedded network by connecting to the network and starting the discovery process. During this process, all nodes report their configuration, the installed services and instances and other management related information to the Planning and Management tool. The in-network storage of management information ensures that the system model is always up-to-date, what cannot be guaranteed if the system model is kept at an external connection.

5.2 Stub/Skeleton and eSDL Generation

Development tools for Web services typically support two different Web service creation processes: an interface-first approach and an implementation-first approach. Using the interface-first approach a developer specifies a Web service using WSDL and the development tool automatically create a service stub containing suitable interfaces that can be filled with the application logic by the developer. The implementation-first approach works the other way round. The developer implements the Web service using a programming language and the Web service development tool automatically creates a corresponding WSDL document (perhaps using some annotation mechanism that allows specifying which methods should be callable via Web service interfaces and which not). The same mechanisms can be provided for embedded services, too. The ϵSOA platform contains a (basic) implementation for both kinds of tools to showcase this functionality.

5.3 Monitoring and Configuration

The Planning and Management tool can also be used for basic monitoring tasks. It evaluates the keep-alive signals sent by each node and automatically detects failed nodes. Failed nodes can be displayed with a special icon in the graphical overview of the network. Additionally, this view can be colorized based on the metrics calculated by the placement optimizer, what allows a quick comparison of different placement alternatives. A possible extension is to extend the coloring to sensor values, too. In this case, the Management and Planning tool automatically subscribes to all sensors providing a specific measurement and displays the received values.

5.4 Summary and Ongoing Work

In this section, we gave a short overview over the tool support provided by the ϵSOA platform. Development tools are a crucial part for embedded networks. Automation systems are too complex to be built and managed by single developers. Instead a multitude of different actors, probably from different companies, contribute to the development of automation systems. Each of these actors has to be supported with tools that allow the actor to efficiently perform his tasks and at the same time ensure the correctness of the overall system. In the ϵSOA platform, the central junction point where all information is aggregated is the system model. Different actors can contribute information to the model, either directly or indirectly via information pre-installed on nodes in the network. The ϵSOA platform provides a set of tools for

- the automated generation of service stubs and service description documents
- the installation and management of services and service libraries
- the definition of the hardware model, including node and link characteristics
- the composition of service to applications
- the monitoring and reconfiguration of installed systems

The ϵSOA platform currently offers no tool support for the Installer, i.e., the person that installs nodes and performs the basic configuration. The Installer is responsible for adding installation specific information to the node and service descriptions, e.g., the location information. This task can be well supported by a Web based tool, which automatically stores all entered information in the system model and the description documents on the nodes.

Another tool we are working on is a pattern development tool. This tool could not only provide a graphical interface for creating patterns, but also include some algorithms for a semi-automatic generation of patterns based on an analysis of already installed applications. These ideas are presented in more detail at the end of this thesis.

We are also working on an Eclipse based version of the tools presented in this section. The goal is to achieve a seamless integration of the code development tools required for the implementation of individual services and the modeling/composition tools required for the creation of whole networks.

CHAPTER 6

Prototypes

The prototypes presented in this section are implemented using the TMote Sky / TelosB platform with a MSP430 microcontroller with 48 kilobytes of program memory and 10 kilobytes of RAM. The nodes are running a TinyOS based (Smart Home Prototype) and a Contiki based (Lighting Prototype) implementation of the ϵSOA platform. The visualization/configuration front ends are realized with a Java based implementation of the platform.

6.1 Smart Home Prototype

Increasing resource prices and ecological considerations are creating the need for smarter power grids. Besides the mere distribution of energy to private, commercial and industrial sites, future power grids also have to be able to manage the distributed production and consumption of energy. The widespread use of renewable energies, e.g., through solar cells, requires the coordination of hundreds (or thousands) of small energy producers instead of a few large power plants we see in current power grids.

The second key challenge for future power grids will be the incorporation of electrically powered cars. These cars present both a challenge and an opportunity for the owner of a power grid. On the one hand, large amounts of energy have to be provided for charging the batteries of these cars; on the other hand these batteries can also contribute to the stability of the power grid by providing massive storage capabilities that can be used to soften load peaks. These storage capacities also play an important role for the efficient usage of renewable energies. Because renewable energy cannot be produced on-demand, large storage capacities are required if a substantial amount of the overall power consumption should be generated out of regenerative resources. Otherwise a lot of the produced energy will be wasted because it is not needed at the time it is produced, an effect that is already observable for some wind energy production sites. By using the batteries of electrically powered cars as energy buffers in these cases, one can greatly increase the efficiency of regenerative power sources.

To decrease the per head energy consumption of the population, smart energy management has to be employed not only in the power grid itself, but also inside the households. A possible scenario is to provide power at varying prices throughout the day, based on the current power demand. A smart energy meter inside the household will supply the power price to intelligent consumers that will optimize their power consumption in order to minimize the overall power costs. Possible examples are a refrigerator that will avoid cooling during load peaks, or a smart washing machine which will start washing when power prices are cheap. Such price mechanisms allows building energy markets that will automatically regulate the power consumption and can be an important building block of stable, decentralized power grids.

A central building block of any of these systems is an efficient, scalable and secure information system. Data has to be acquired, aggregated and stored at thousands of consumers and producers distributed throughout the power grid. This requires highly scalable and reliable information systems that at the same time provide a real-time overview of the state of the power grid to allow regulating the flow of energy through the grid. We call such a combined power grid and computer network an InfoGrid. The development of such an InfoGrid is an important milestone towards next generation power grids.

The Smart Home Demonstrator is used to showcase a scenario with a smart metering device and varying energy prices. Based on the ϵSOA platform, we developed a demonstrator, which covers a future home automation scenario. The assembling of our demonstrator is shown in Figure 6.1. We assume that in the near future energy providers use dynamically changing energy prices in order to influence the overall energy consumption in a way that smoothes load peaks. We further assume that some kind of power storage system, such as the battery of an electric car, is present in future homes. We implemented the following scenario: A household comprising a battery and loads (a refrigerator and 2 lights) with different power consumption and energy saving options. One task of the automation logic is to minimize the energy costs throughout the day. If prices are cheap, the battery is charged and the refrigerator cools down to a lower threshold. If prices are high, the house is disconnected from the power grid and draws its energy from the battery. Additionally, the refrigerator is put to energy saving mode, i.e., it stops cooling until an upper temperature threshold is reached. There are other functional requirements not presented in detail here, e.g., the home has to connect to the power grid if the summed consumption of all devices exceeds the power of the battery, the battery should not be completely depleted, etc. The electricity prices are delivered by an external Web service, which is represented by a virtual sensor in the network. The used ZigBee based motes possess a set of I/O devices used to read signals from the switches and turn on or off the loads. The requirement to support end user programming can be also motivated by this example. Starting from a traditional control system, the

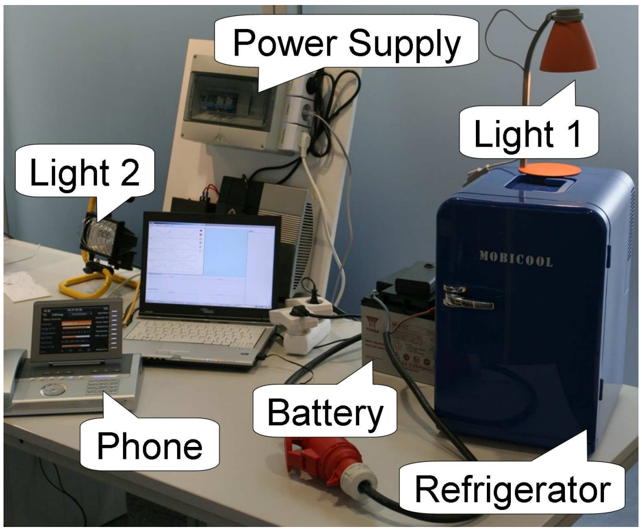


Figure 6.1: Smart Home Demonstrator

user can add a battery and install a new pattern to benefit from the described price saving mechanism. Furthermore, the ϵSOA approach offers a new flexibility to end users. Changes, e.g. of the lighting, can be performed easily using the graphical user interface provided by the Planning and Management tool of the ϵSOA platform. The demonstrator also points out how different devices can be used to administrate the embedded network. The programmable phone, for instance, can be used to monitor sensor readings and to adjust thresholds, such as the maximum temperature of the fridge.

6.2 Intelligent Lighting Prototype

The lighting demonstrator shows the failure recovery mechanisims available in the ϵSOA platform and demonstrates that fast reaction times to sensor events can be realized in distributed control systems. Figure 6.2 shows the demonstrator setup. The demonstrator comprises 12 motes. The lights are grouped into four groups. One group is located at the desk and comprises four lamps, which are controlled by four separate motes II, III, VI and VII. The second group is the meeting area, which comprises two lamps and two corresponding motes, Mote $VIII$ and XII. These two groups also contain a combined brightness/activity sensor, which is attached to Mote V and X, respectively. This sensor can be used to dim the lights in the corresponding area according to the current daylight and to turn lights on or off whenever a human enters or leaves the observed area. Mote IV and XI each control a group of smaller lights. The lights attached to Mote XI provide general illumination for the room. The wall near Mote IV contains a whiteboard and can be used as projection area for a digital light projector. The wall can be illuminated

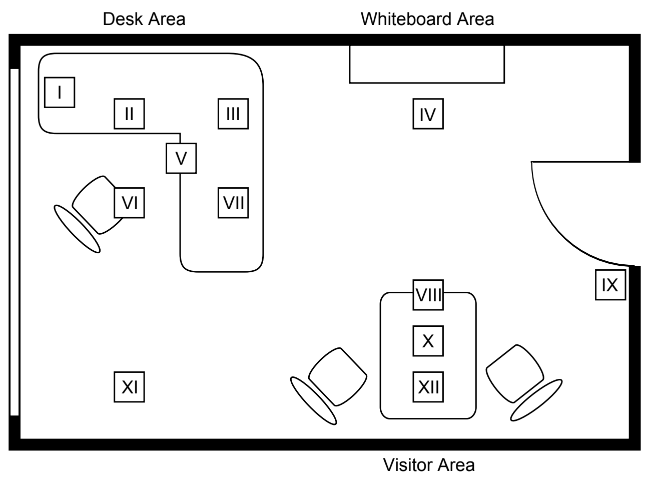


Figure 6.2: Intelligent Lighting Prototype

with a set of 3 lights, which are controlled by Mote IV. Mote I and IX are attached to switches. Mote IX is the master switch that allows to turn off all lights in the room and to activate a "maintenance mode" in which all lights are turned on. The switch located at the desk, I, can be used to switch the light at the whiteboard on or off, and to turn on a "beamer mode". In the beamer mode, the lights at the whiteboard and at the visitor area are turned off.

All three failure recovery mechanisms described in Section 4.1 are implemented in the demonstrator.

Implicit Compensation If a lamp fails or does not provide the expected level of brightness due to aging, the control logic will automatically adjust the brightness of the remaining lamps in order to achieve the desired brightness level. If the target level cannot be reached, this situation can be reported via an email interface. The email notification has two purposes. It can be used to trigger a proactive replacement of aged lamps before an actual failure occurs and ensures that malfunctioning lamps are reported immediately and can be replaced in a timely fashion.

Graceful Degradation If the presence/activity sensor (or the corresponding mote) is missing or fails, the light control logic switches to a graceful degradation mode. In this mode, the dimming of lights is turned off (if no brightness information is available) and/or the light is always turned on when the master switch is turned on (if no activity information is available). When the sensor becomes available again, the control application automatically switches back to normal operation. Additionally, the failure of the sensor is reported via the email notification interface.

Redundancy In order to allow a compensation of node failures, the light logic for the desk and the visitor area are installed redundantly on the nodes in each group.

Such a configuration can be created by using an application pattern with the redundancy combination strategy in the system model, as introduced in Section 2.5. If the node currently executing the control logic fails, one of the remaining nodes automatically takes over control of the group. This is done by changing the configuration of the Stream Routers to send data to the replacement node instead of the failed node. When the failed node is online again, control is automatically handed back and the original configuration is restored. Node failures are also reported through the email notification interface.

The communication in the lighting demonstrator is a mixture of periodic transmissions from the brightness sensors and event driven communication whenever a switch is pressed or activity in one of the areas is detected. A challenge in the demonstrator is that the event driven communication is very bursty and has to be very fast to provide quick response times for the user of the system. The burstiness is created by the large number of individual lamps, which have to be switched simultaneously. Assume the user enters the room and hits the master switch. In this case, the control logic of each of the four light groups has to be updated with the new state of the master switch. The control logic then calculates the new dim value for the lights in the group and sends this information to all other motes in the corresponding group (three in the desk group and one in the visitor group if the logic runs on one of the lights in this group). The overall execution time for the execution of these service chains is clearly dominated by the communication costs for the transmission of data between the involved services. The demonstrator uses UDP based communication and achieves end-to-end response time of less than 0.4 seconds, including proactive retransmission to compensate packet losses. Better response times can be achieved if a faster communication technology is used.

The lighting demonstrator shows the failure recovery mechanisms available in the ϵSOA platform and a model-first development approach. Based on the system model developed in the engineering tool, service stubs for the implementation of the individual services can be generated. Based on these implementations and the system model, deployable images for each node, including all configuration information, are automatically generated by the development tools. To show the failure compensation mechanisms, the brightness/activity sensor is modeled as optional component and the control logic automatically reacts to the failure of this sensor by switching to a degradation mode. To allow a compensation of failed nodes, the light control logic is installed redundantly on several nodes by specifying a corresponding combination strategy in the system model. A failure or an increased brightness of single lights can also be compensated by the control logic, which automatically adjusts the brightness of the remaining lights. All failures in the demonstrator are reported via email.

6.3 Related Work

We already presented related work for individual topics at the end of each section. We will therefore focus on appraches that also propose an overall system architecture in this section.

Embedded Virtual Machines The embedded virtual machines proposed by Mangharam et al.[96, 97] provide an abstraction layer that allows distributing control tasks over multiple nodes in the network. Through the installation of multiple copies of (parts) of the control task, failures can be compensated - a mechanism that is comparable to the failure compensation used in the ϵSOA platform.

ITEA - SIRENA/SODA/SOCRADES A research project with a scope similar to the work presented here is SOCRADES[1], which started parallel to the work presented here. SOCRADES leverages work conducted in the SODA[2] and SIRENA[3] projects. SOCRADES[13, 140] also uses a service oriented paradigm to model and implement automation systems. One result of the SOCRADES project is a demonstrator that shows that SOA is a feasible technology for implementing distributed automation systems. The SOCRADES platform and the demonstrator are realized based on an implementation of the DPWS stack. The DPWS implementation requires fairly powerful nodes compared to the microcontroller based platform presented in this work. By using binary XML technologies and the optimized code generation presented in this work, the resource requirements for executing distributed control tasks can be reduced. The SOCRADES project has a strong focus on the integration of field level devices with enterprise systems, such as SAP. De Souza et al.[22] propose an architecture for such an integrated system and show how field level devices and ERP systems can be linked together. The key enabling technology for this integration - Web service based interfaces - is also provided by the ϵSOA platform. In our demonstrator we showed how an integration of ϵSOA and Web services is possible. This work could be extended to achieve an integration with enterprise systems, like the integration shown in the SOCRADES project. The ϵSOA platform offers some features that are not available in the SOCRADES platform. The three main features are (1) the optimized placement of services (including the possibility to move services between nodes at runtime), (2) the failure recovery mechanism using preconfigured redundancy, and (3) the pattern based service composition. These concepts can be transferred to other service oriented development platforms and can be used to ease the engineering process and to build self-optimizing and self-healing systems.

[1]http://www.socrades.eu
[2]http://www.soda-itea.org/
[3]http://www.sirena-itea.org/

Grid Platforms (C3-Grid) An interesting observation is that the execution of requests/applications in some grid platforms, such as the C3-Grid[4][47, 80], shows similarities with the concepts presented in this thesis. The C3-Grid is targeted at collaborative climate data processing and also uses a service oriented system architecture. A typical processing task in such a grid is to gather data from several sources, perform some preprocessing on this data, join/aggregate the collected data and finally create an output that is submitted to the requester. The individual parts of the processing task can be executed on different nodes in the grid, which also leads to an optimization problem comparable to the service placement problem analyzed in this work[5]. Of course the individual challenges encountered in both domains are different, e.g., the resource limitations in embedded networks or the heterogeneity of data sources in a grid environment. Nevertheless, the architectural similarities are an interesting starting point for thinking about an integration of grids and embedded networks, e.g., to source out processing intensive evaluation tasks from the embedded network to the grid platform, or enhance the grid platform with direct access to field devices for improved data freshness.

[4]`http://www.c3grid.de`

[5]A difference is that the service placement problem in embedded networks is more focused on optimizing the communication between nodes, whereas grid applications are often trying to optimize the utilization of the nodes in the grid.

CHAPTER 7

Summary & Future Work

In this work we presented the ϵSOA development platform. The ϵSOA platform is targeted at the development of control and automation applications in distributed heterogeneous embedded networks. It is based on three core design principles: a service oriented system architecture, a model driven development paradigm and a stream based execution model. The combination of these principles allows creating highly customizable systems that can optimize their behavior based on application requirements and network characteristics. The implementation of the aforementioned concepts on resource constrained microcontrollers requires new concepts and/or the adaptation of existing solutions, such as Web service technologies. Besides the conceptual design of the platform, this work also describes an execution environment and a corresponding development toolsuite for the design, installation and management of control applications in resource constrained environments. The feasibility of the solution was demonstrated with prototypical implementations in the building automation domain.

The concepts and the implementation described in this work are intended to form the basic building blocks for future, SOA based automation systems. There are several directions for future research. Many of these were already presented at the end of the individual sections. Besides these topics, there are some additional interesting research directions, which we will outline in the following paragraphs.

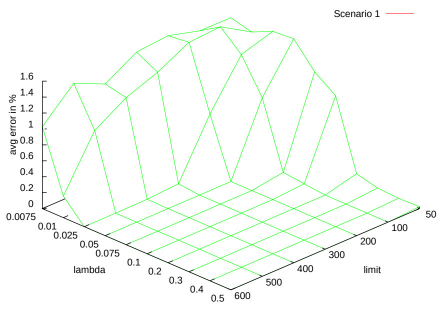
Semantic Control Throughout this work, we showed some example applications in the area of building automation. These examples were mostly based on state-of-the-art automation tasks, which can also be realized with nowadays automation solutions. Embedded networks possess a huge potential beyond these comparatively simple automation tasks. The seamless access to devices in our surroundings and the possibility to dynamically add nodes to the control network offer unpreceded interaction possibilities between devices. One scenario are smart mobile devices which influence their surroundings to guarantee an optimal user experience. Consider for example a digital light projector (DLP), which should be automatically integrated into the light control system when it is brought to a room. In most cases it is practically infeasible to pre-plan the addition of all such devices during the creation of the light control system. An interesting vision is an automated integration of new devices based on semantic information that also expresses certain environmental conditions (e.g. a maximum brightness specified by the DLP). This concept could be extended to the components used inside the control application, too. New sensor/actuator devices could describe their capabilities (and associated costs) and the automation logic could dynamically select the most appropriate device(s) for achieving a certain domain goal. Many of these concepts are studied in the context of agent based systems, however these typically lack an implementation on resource constrained devices. An interesting idea is the combination of such agent based approaches and the model driven paradigm used in the ϵSOA platform in order to combine the flexibility and the resource efficiency provided by the different approaches.

Pattern Learning / Automated Service Composition In the current implementation of the ϵSOA platform, application patterns are manually designed by the developer. An interesting topic for future research is the automatic learning of such patterns from a repository of installed applications. These patterns could not only cover concrete applications (e.g. a pattern for a heating, ventilation, and air-condition (HVAC) application), but also meta-patterns such as a sensor-logic-actuator chain. This information can be used to improve the tooling support for the creation of patterns by providing additional hints for the developer that concern structural aspects of the created pattern. Another application field of such a pattern recognition approach is an automated service composition without the explicit specification of patterns. This could be used to provide a “smart” copy&paste functionality that allows to copy applications between installations, even if both applications do not have identical components but only components with similar functionality.

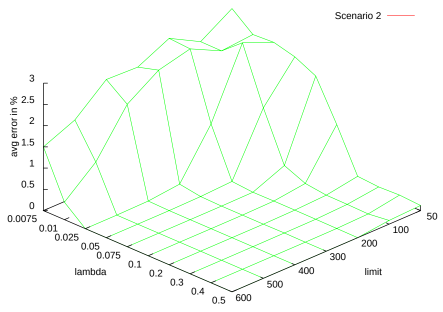
Timing Constraints In some application fields of embedded networks, such as discrete manufacturing systems, timing constraints play an important role. One aspect is the reaction speed, i.e., the latency between a measurement at a sensor device and the corresponding reaction of an actuator device, which typically has to stay below a certain threshold. A second aspect is determinism. In many cases it is decisive to keep the jitter, i.e., the variation of this latency, low. In nowadays automation systems, timing constraints are often implicitly guaranteed through the cyclic execution model. Based on the maximum execution time of a cycle, an upper bound for the execution time of the automation logic can be derived. This model is problematic if additional functionality should be added to an existing system. Furthermore, this model of execution cannot be applied in distributed automation systems. Using a model driven development approach, such requirements could be specified at the application level as end-to-end requirements. Wiklander et al.[167] propose such an approach for components on a single embedded controller. In the ϵSOA platform, a solution for a distributed execution of applications on multiple internetworked devices has to be developed, e.g., an optimizer component that breaks down end-to-end requirements for individual links and service invocations and derives schedules for TDMA communication schemes and prioritizes/schedules service invocations on the nodes.

APPENDIX A

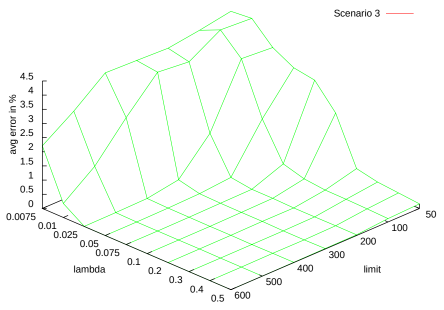
Service Placement Benchmarks



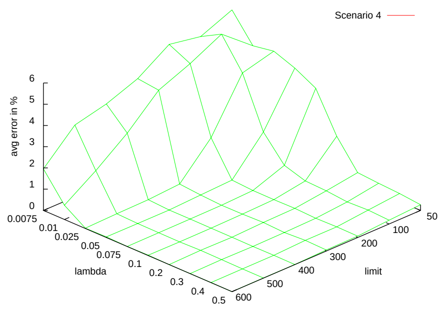
(a) Benchmark Scenario 1



(b) Benchmark Scenario 2

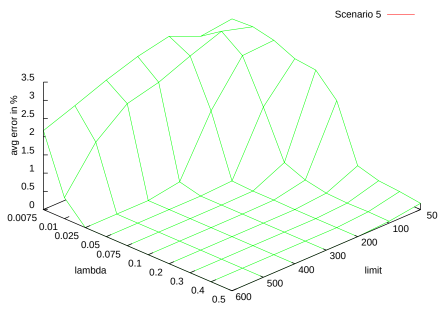


(c) Benchmark Scenario 3

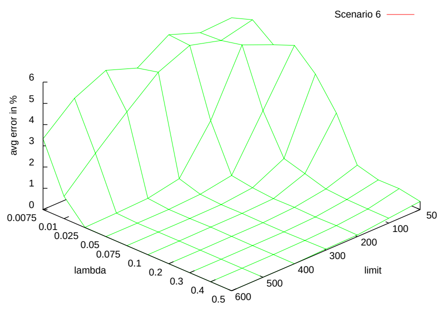


(d) Benchmark Scenario 4

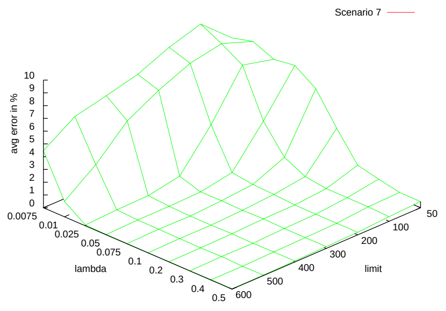
Figure A.1: Lambda / Limit Test Series



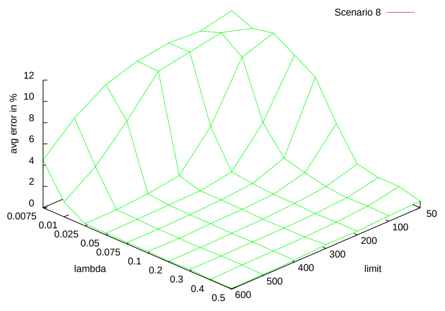
(e) Benchmark Scenario 5



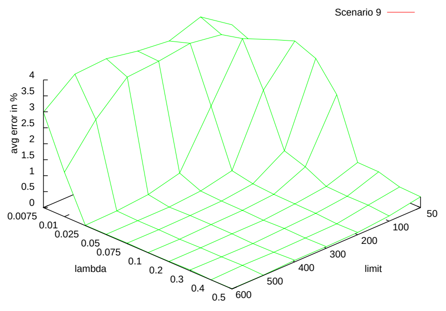
(f) Benchmark Scenario 6



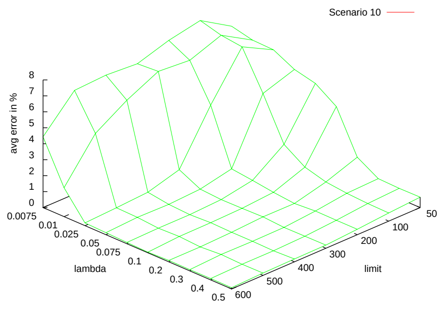
(g) Benchmark Scenario 7



(h) Benchmark Scenario 8

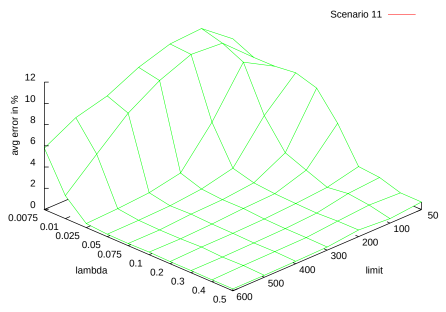


(i) Benchmark Scenario 9

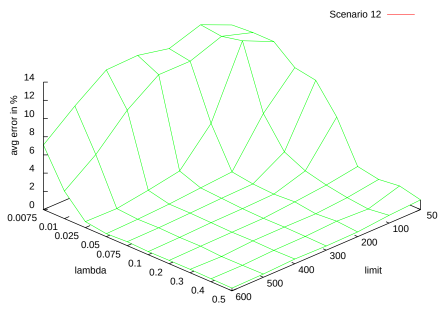


(j) Benchmark Scenario 10

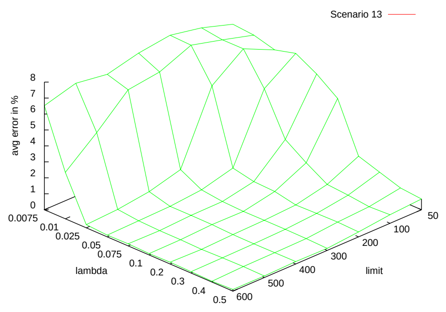
Figure A.1: Lambda / Limit Test Series



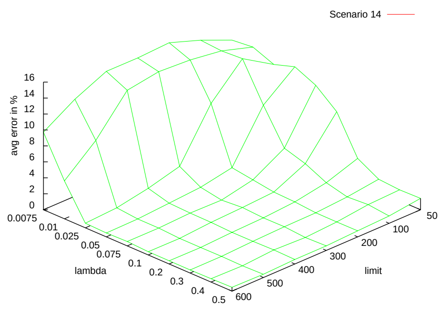
(k) Benchmark Scenario 11



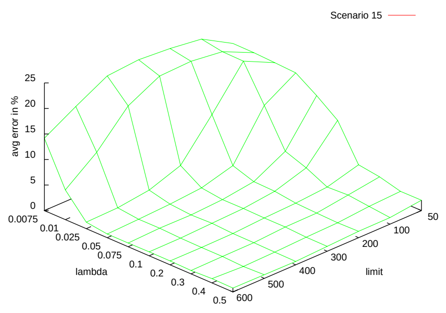
(l) Benchmark Scenario 12



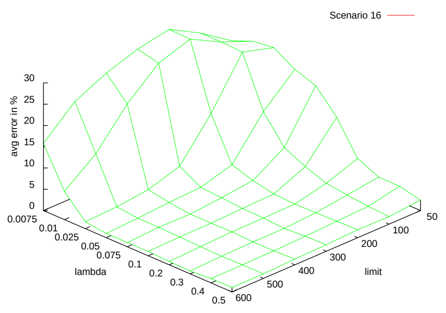
(m) Benchmark Scenario 13



(n) Benchmark Scenario 14

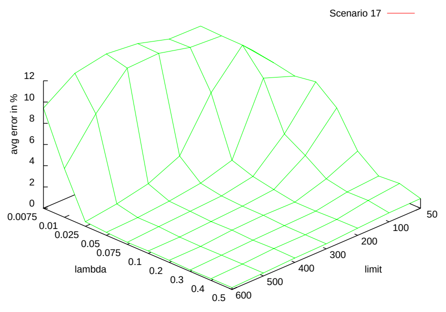


(o) Benchmark Scenario 15

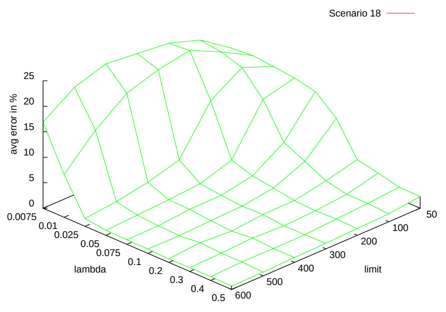


(p) Benchmark Scenario 16

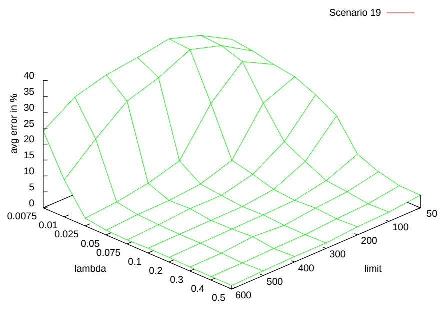
Figure A.1: Lambda / Limit Test Series



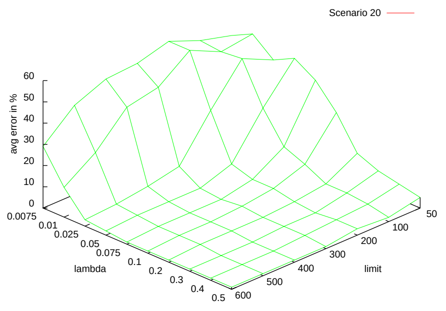
(q) Benchmark Scenario 17



(r) Benchmark Scenario 18



(s) Benchmark Scenario 19



(t) Benchmark Scenario 20

Figure A.1: Lambda / Limit Test Series

APPENDIX B

XML Schema Definitions

B.1 eHDL XML Schema Definition

The information model for the embedded Hardware Description Language (eHDL) is introduced in Section 2.3.1. A corresponding XML based notation is described in Section 3.3. The XML Schema definition for the XML notation can be found in Listing B.1

```
<schema targetNamespace="http://in.tum.de/eSOA/hardware"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:tns="http://in.tum.de/eSOA/hardware">

  <element name="hardware" type="tns:HardwareType"></element>

  <!-- hardware description container -->
  <complexType name="HardwareType">
    <sequence>
      <!-- list of resources -->
      <element name="resource" type="tns:ResourceType"
               maxOccurs="unbounded" minOccurs="0">
      </element>
      <!-- reference to hardware definition -->
      <element name="hardware" type="long"
               maxOccurs="unbounded" minOccurs="0">
      </element>
      <!-- list of properties -->
      <element name="property" type="tns:PropertyType"
               maxOccurs="unbounded" minOccurs="0">
      </element>
    </sequence>
  </complexType>
  <!-- property -->
  <complexType name="PropertyType">
    <attribute name="name">
      <simpleType>
        <restriction base="string">
          <enumeration value="Floor"></enumeration>
          <enumeration value="Room"></enumeration>
          <!-- ... -->
        </restriction>
      </simpleType>
    </attribute>
    <attribute name="value" type="string"></attribute>
  </complexType>


  <!-- resource definition -->
  <complexType name="ResourceType">
    <attribute name="name">
      <simpleType>
        <restriction base="string">
          <enumeration value="RAM"></enumeration>
          <enumeration value="ProgramMemory"></enumeration>
          <enumeration value="Flash"></enumeration>
          <!-- ... -->
        </restriction>
      </simpleType>
    </attribute>
    <attribute name="value" type="int"></attribute>
  </complexType>
</schema>
```

Listing B.1: eHDL XML Schema Definition

B.2 eSDL XML Schema Definition

The information model for the embedded Service Description Language (eSDL) is introduced in Section 2.3.2 and a corresponding XML notation in Section 3.4. The XML Schema definition for the XML notation can be found in Listing B.2

```
<schema targetNamespace="http://in.tum.de/eSOA/service"
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://in.tum.de/eSOA/service">
  <complexType name="ServiceType">
    <sequence>
      <element name="property" type="tns:PropertyType"
               maxOccurs="unbounded" minOccurs="0">
      </element>
      <element name="operation" type="tns:OperationType"
               maxOccurs="unbounded" minOccurs="0">
      </element>
    </sequence>
    <attribute name="name" type="string" use="required">
    </attribute>
  </complexType>

  <element name="service" type="tns:ServiceType">
  </element>

  <complexType name="OperationType">
    <sequence>
      <element name="input" type="tns:MessageType"
               maxOccurs="1" minOccurs="0">
      </element>
      <element name="output" type="tns:MessageType"
               maxOccurs="1" minOccurs="0">
      </element>
    </sequence>
    <attribute name="address" type="byte" use="required">
    </attribute>
  </complexType>

  <complexType name="ParameterType">
    <sequence>
      <element name="property" type="tns:PropertyType"
               maxOccurs="unbounded" minOccurs="0">
      </element>
    </sequence>
    <attribute name="datatype" type="tns:DatatypeType" use="required">
    </attribute>
    <attribute name="measurandtype" type="tns:MeasurandtypeType"
               use="required">
    </attribute>
  </complexType>

  <complexType name="PropertyType">
    <attribute name="name" type="tns:PropertyName">
    </attribute>
    <attribute name="value" type="string">
    </attribute>
  </complexType>


  <simpleType name="PropertyName">
    <restriction base="string">
      <enumeration value="Unit"></enumeration>
      <enumeration value="Precision"></enumeration>
      <!-- ... and so on ... -->
    </restriction>
  </simpleType>

  <simpleType name="DatatypeType">
    <restriction base="string">
      <enumeration value="string"></enumeration>
      <enumeration value="int"></enumeration>
      <enumeration value="short"></enumeration>
      <enumeration value="byte"></enumeration>
      <enumeration value="boolean"></enumeration>
      <!-- ... and so on ... -->
    </restriction>
  </simpleType>

  <simpleType name="MeasurandtypeType">
    <restriction base="string">
      <enumeration value="Temperature"></enumeration>
      <enumeration value="Light"></enumeration>
      <enumeration value="Humidity"></enumeration>
      <!-- ... and so on ... -->
    </restriction>
  </simpleType>

  <complexType name="MessageType">
    <sequence>
      <element name="parameter" type="tns:ParameterType"
               maxOccurs="unbounded" minOccurs="0">
      </element>
    </sequence>
  </complexType>
</schema>
```

Listing B.2: eSDL XML Schema Definition

B.3 eSCL XML Schema Definition

The information model for the embedded Service Choreography Language (eSCL) and a corresponding XML notation are introduced in Section 3.5. The XML Schema definition for the XML notation can be found in Listing B.3

```
<schema targetNamespace="http://in.tum.de/eSOA/choreography"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:escl="http://in.tum.de/eSOA/choreography">
  <element name="choreography" type="escl:ChoreographyType"></element>

  <!-- choreography container -->
  <complexType name="ChoreographyType">
    <sequence>
      <element name="instances" type="escl:InstancesType" />
      <element name="operationGroups" type="escl:OperationGroupsType" />
      <element name="dataStreams" type="escl:DataStreamsType" />
      <element name="streamGroups" type="escl:StreamGroupsType" />
    </sequence>
  </complexType>

  <!-- list of involved instances -->
  <complexType name="InstancesType">
    <sequence>
      <element name="instance" type="escl:InstanceType"
               maxOccurs="unbounded" minOccurs="0" />
    </sequence>
  </complexType>

  <!-- instance definition -->
  <complexType name="InstanceType">
    <!-- globally unique instance identifier -->
    <attribute name="instanceId" type="unsignedShort" use="required" />
    <!-- reference to service desctription -->
    <attribute name="service" type="string" use="required" />
  </complexType>

  <!-- list of operation groups -->
  <complexType name="OperationGroupsType">
    <sequence>
      <element name="operationGroup" type="escl:OperationGroupType"
               maxOccurs="unbounded" minOccurs="0" />
    </sequence>
  </complexType>

  <!-- operation group definition (list of redundant operations) -->
  <complexType name="OperationGroupType">
    <attribute name="id" type="unsignedByte" use="required" />
    <sequence>
      <element name="operation" type="escl:OperationType"
               maxOccurs="unbounded" minOccurs="0" />
    </sequence>
  </complexType>
  <!-- operation reference -->
  <complexType name="OperationType">
    <attribute name="instanceId" type="unsignedShort" use="required" />
    <attribute name="operation" type="unsignedByte" use="required" />
  </complexType>

  <!-- list of data streams -->
  <complexType name="DataStreamsType">
    <sequence>
      <element name="dataStream" type="escl:DataStreamType"
               maxOccurs="unbounded" minOccurs="0" />
    </sequence>
  </complexType>

  <!-- data stream definition -->
  <complexType name="DataStreamType">
    <attribute name="source" type="unsignedByte" use="required" />
    <attribute name="drain" type="unsignedByte" use="required" />
    <attribute name="streamId" type="unsignedByte" use="required" />
  </complexType>

  <!-- list of stream groups -->
  <complexType name="StreamGroupsType">
    <sequence>
      <element name="streamGroup" type="escl:StreamGroupType"
               maxOccurs="unbounded" minOccurs="0" />
    </sequence>
  </complexType>

  <!-- stream group definition -->
  <complexType name="StreamGroupType">
    <attribute name="id" type="unsignedByte" use="required" />
    <sequence>
      <element name="dataStream" type="escl:DataStreamTypeRef"
               maxOccurs="unbounded" minOccurs="0" />
    </sequence>
  </complexType>

  <!-- reference to datastream -->
  <complexType name="DataStreamTypeRef">
    <attribute name="streamId" type="unsignedByte" use="required" />
  </complexType>
</schema>
```

Listing B.3: eSCL XML Schema Definition

B.4 Meta-Information Repository: XML Schema Definition

The Meta-Information Repository is presented in Section 3.8. It defines a XML message format for the storage of metadata entries. The corresponding XML Schema definition can be found in Listing B.4

```
<schema targetNamespace="http://www3.in.tum.de/mir"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:mir="http://www3.in.tum.de/mir">

  <element name="entry" type="mir:EntryType"></element>

  <complexType name="EntryType">
    <sequence>
      <!-- namespace is inhereted if not specified here -->
      <element name="namespace" type="string"
               maxOccurs="unbounded" minOccurs="0"></element>
      <!-- identifier of the entry -->
      <element name="identifier" type="string"></element>
      <!-- reference to parent identifier -->
      <element name="parent" type="string"></element>
      <!-- numeric value for identifier -->
      <element name="position" type="int"></element>
      <!-- type, not all entries shown here -->
      <element name="type">
        <simpleType>
          <restriction base="string">
            <enumeration value="Object"></enumeration>
            <enumeration value="List"></enumeration>
            <enumeration value="Subtype"></enumeration>
            <enumeration value="string"></enumeration>
            <enumeration value="byte"></enumeration>
            <enumeration value="short"></enumeration>
            <enumeration value="int"></enumeration>
            ...
          </restriction>
        </simpleType>
      </element>
      <!-- access rights, see MIB -->
      <element name="access" maxOccurs="1" minOccurs="0">
        <simpleType>
          <restriction base="string">
            <enumeration value="read-write"></enumeration>
            <enumeration value="read-only"></enumeration>
            <enumeration value="write-only"></enumeration>
            <enumeration value="not-accessible"></enumeration>
          </restriction>
        </simpleType>
      </element>
      <!-- status of entry, see MIB -->
      <element name="status" maxOccurs="1" minOccurs="0">
        <simpleType>
          <restriction base="string">
            <enumeration value="mandatory"></enumeration>
            <enumeration value="optional"></enumeration>
            <enumeration value="obsolete"></enumeration>
            <enumeration value="deprecated"></enumeration>
          </restriction>
        </simpleType>
      </element>
      <!-- optional human readable description, see MIB -->
      <element name="description" type="string"
               maxOccurs="1" minOccurs="0"></element>
      <!-- optional reference, see MIB -->
      <element name="reference" type="string"
               maxOccurs="unbounded" minOccurs="0"></element>
      <!-- optional default value, see MIB -->
      <element name="default" type="string"
               maxOccurs="unbounded" minOccurs="0"></element>
    </sequence>
  </complexType>
</schema>
```

Listing B.4: Meta-Information Repository XML Schema Definition

Bibliography

[1] bzip2. http://www.bzip.org/.

[2] Electronic device description language (EDDL). http://www.eddl.org.

[3] PROFIBUS. http://www.profibus.com/.

[4] ISO/IEC 7498-1: Open systems interconnection – basic reference model, 1994.

[5] G. A. Agha. Actors: A model of concurrent computation in distributed systems. In *The MIT Press Series in Artificial Intelligence*. MIT Press, Cambridge, 1986.

[6] D. Balfanz, G. Durfee, J. Staddon, and J. Staddon. Efficient tracing of failed nodes in sensor networks. In *In Proceedings of the First ACM International Workshop on Wireless Sensor Networks and Applications*, pages 122–130, 2002.

[7] W. Berrayana, G. Pujolle, and H. Youssef. Xlengine: a cross-layer autonomic architecture with network wide knowledge for qos support in wireless networks. In *IWCMC '09: Proceedings of the 2009 International Conference on Wireless Communications and Mobile Computing*, pages 170–175, New York, NY, USA, 2009. ACM.

[8] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 169–179, New York, NY, USA, 2007. ACM.

[9] A. Bucchiarone and S. Gnesi. A survey on services composition languages and models. In A. Bertolino and A. Polini, editors, *in Proceedings of International Workshop on Web Services Modeling and Testing (WS-MaTe2006)*, pages 51–63, Palermo, Sicily, ITALY, June 9th 2006.

[10] C. Buckl, S. Sommer, A. Scholz, A. Knoll, and A. Kemper. Generating a tailored middleware for wireless sensor network applications. In *Proceedings of the 2008 IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (sutc 2008)*, pages 162–169, Washington, DC, USA, 2008. IEEE Computer Society.

[11] C. Buckl, S. Sommer, A. Scholz, A. Knoll, A. Kemper, J. Heuer, and A. Schmitt. Services to the field: An approach for resource constrained sensor/actor networks. *Advanced Information Networking and Applications Workshops, International Conference on*, 0:476–481, 2009.

[12] M. E. Cambronero, G. Díaz, E. Martínez, and V. Valero. A comparative study between wsci, ws-cdl, and owl-s. In *ICEBE '09: Proceedings of the 2009 IEEE International Conference on e-Business Engineering*, pages 377–382, Washington, DC, USA, 2009. IEEE Computer Society.

[13] A. Cannata, M. Gerosa, and M. Taisch. Socrades: A framework for developing intelligent systems in manufacturing. In *Industrial Engineering and Engineering Management, 2008. IEEM 2008. IEEE International Conference on*, pages 1904 –1908, 8-11 2008.

[14] V. Cerny. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45:41–51, 1985. 10.1007/BF00940812.

[15] Y. chee Tseng, S. po Kuo, H. wei Lee, and C. fu Huang. Location tracking in a wireless sensor network by mobile agents and its data fusion strategies. In *The Computer Journal*, pages 448–460, 2003.

[16] M. Chen, S. Gonzalez, and V. Leung. Applications and design issues for mobile agents in wireless sensor networks. *Wireless Communications, IEEE*, 14(6):20 –26, december 2007.

[17] M.-J. Choi, H.-M. Choi, J. Hong, and H.-T. Ju. Xml-based configuration management for ip network devices. *Communications Magazine, IEEE*, 42(7):84 – 91, july 2004.

[18] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[19] B. Cohen. The bittorrent protocol specification. http://www.bittorrent.org/beps/bep_0003.html, 2008.

[20] A. Colorni, M. Dorigo, and V. Maniezzo. Distributed optimization by ant colonies. In *ECAL'91: European Conference on Artificial Life*, 91.

[21] M. Conti, G. Maselli, G. Turi, and S. Giordano. Cross-layering in mobile ad hoc network design. *Computer*, 37:48–51, 2004.

[22] L. M. S. de Souza, P. Spiess, D. Guinard, M. Köhler, S. Karnouskos, and D. Savio. Socrades: A web service based shop floor integration infrastructure. In C. Floerkemeier, M. Langheinrich, E. Fleisch, F. Mattern, and S. E. Sarma, editors, *IOT*, volume 4952 of *Lecture Notes in Computer Science*, pages 50–67. Springer, 2008.

[23] B. Deb, S. Bhatnagar, and B. Nath. A topology discovery algorithm for sensor networks with applications to network management. In *IEEE CAS Workshop*, 2002.

[24] B. Deb, S. Bhatnagar, and B. Nath. Stream: Sensor topology retrieval at multiple resolutions. *Kluwer Journal of Telecommunications Systems*, 26:285–320, 2003.

[25] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimisation: Nsga-ii. In *Proceedings of the 6th International Conference on Parallel Problem Solving from Nature*, PPSN VI, pages 849–858, London, UK, 2000. Springer-Verlag.

[26] G. Decker, H. Overdick, and J. M. Zaha. J.m.: On the suitability of ws-cdl for choreography modeling. In *In: Proceedings of Methoden, Konzepte und Technologien für die Entwicklung von dienstebasierten Informationssystemen (EMISA 2006*, 2006.

[27] DMTF. Common information model (cim) specification. version 2.2. http://www.dmtf.org/standards/documents/CIM/DSP0004.pdf, 1999.

[28] DMTF. CIM operations over http. http://www.dmtf.org/standards/published_documents/DSP0200_1.3.1.pdf, 2009.

[29] DMTF. CIM xml document type definition (dtd). http://www.dmtf.org/standards/published_documents/DSP0203_2.3.1.dtd, 2009.

[30] DMTF. Representation of cim in xml, version 2.3.1. http://www.dmtf.org/standards/published_documents/DSP0201_2.3.1.pdf, 2009.

[31] DMTF. Ws-management cim binding specification. http://www.dmtf.org/standards/published_documents/DSP0227_1.0.0.pdf, 2009.

[32] DMTF. CIM schema. http://www.dmtf.org/standards/cim/cim_schema_v2250, 2010.

[33] B. Donnet and T. Friedman. Internet topology discovery: a survey. *IEEE Communications Surveys and Tutorials*, 9(4):2–15, December 2007.

[34] M. Dorigo. *Optimization, Learning and Natural Algorithms.* PhD thesis, Politecnico di Milano, 1992.

[35] EPCglobal. EPCglobal tag data standards, version 1.4, June 2008.

[36] J. Eriksson, A. Dunkels, N. Finne, F. Österlind, and T. Voigt. Mspsim – an extensible simulator for msp430-equipped sensor boards. In *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session*, Delft, The Netherlands, Jan. 2007.

[37] EXIficient. http://exificient.sourceforge.net/.

[38] R. Falk and H.-J. Hof. Security design for industrial sensor networks. *it - Information Technology*, 52(6):331–339, 2010.

[39] R. Farrell and L. Davis. Decentralized discovery of camera network topology. In *Distributed Smart Cameras, 2008. ICDSC 2008. Second ACM/IEEE International Conference on*, pages 1 –10, sept. 2008.

[40] R. Farrell, R. Garcia, D. Lucarelli, A. Terzis, and I.-J. Wang. Localization in multi-modal sensor networks. In *Intelligent Sensors, Sensor Networks and Information, 2007. ISSNIP 2007. 3rd International Conference on*, pages 37 –42, dec. 2007.

[41] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. pages 37–54, 2007.

[42] L. Fredlund. Implementing ws-cdl. In *Proceedings of the second Spanish workshop on Web Technologies (JSWEB 2006)*, Universidade de Santiago de Compostela, November 2006.

[43] K. H. Fritsche. Tinytorrent: Combining bittorrent and sensornets. Master's thesis, University of Dublin, Trinity College, http://hdl.handle.net/2262/850, 2005.

[44] P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. Irisnet: An architecture for a worldwide sensor web. *IEEE Pervasive Computing*, 2:22–33, 2003.

[45] GlassFish. Fast infoset project. https://fi.dev.java.net/.

[46] D. Gmach. *Managing Shared Resource Pools for Enterprise Applications.* PhD thesis, Technische Universität München, 2009.

[47] C. Grimme, T. Langhammer, A. Papaspyrou, and F. Schintke. Negotiation-based choreography of data-intensive applications in the c3grid project. In *German e-Science Conference 2007*, Baden-Baden, May 2007.

[48] M. Guido, Z. Elmar, P. Steffen, G. Frank, T. Dirk, and S. Regina. Devices profile for web services in wireless sensor networks: Adaptations and enhancements. In *IEEE 14th International Conference on Emerging Technologies and Factory Automation (ETFA 2009).* IEEE-ETFA, September 2009. Conference.

[49] R. Gummadi, N. Kothari, T. Millstein, and R. Govindan. Declarative failure recovery for sensor networks. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 173–184, New York, NY, USA, 2007. ACM.

[50] C.-C. Han, R. Kumar, R. Shea, and M. Srivastava. Sensor network software update management: a survey. *Int. J. Netw. Manag.*, 15(4):283–294, 2005.

[51] W. R. Heinzelman, J. Kulik, and H. Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, MobiCom '99, pages 174–185, New York, NY, USA, 1999. ACM.

[52] C. Hewitt. Viewing control structures as patterns of passing messages. *Artif. Intell.*, 8(3):323–364, 1977.

[53] R. Holman, J. Stanley, and T. Ozkan-Haller. Applying video sensor networks to nearshore environment monitoring. *IEEE Pervasive Computing*, 2(4):14–21, 2003.

[54] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94, New York, NY, USA, 2004. ACM.

[55] IEEE. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications (ANSI/IEEE Std 802.11, 1999 Edition (R2003)).* Institute of Electrical and Electronics Engineers, Inc., June 2003.

[56] IEEE. *Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs) (ANSI/IEEE Std 802.15.4).* Institute of Electrical and Electronics Engineers, Inc., June 2006.

[57] IETF. RFC 1213: Management information base for network management of tcp/ip-based internets: Mib-ii.
http://tools.ietf.org/html/rfc1213, 1991.

[58] IETF. RFC 1952: Gzip file format specification version 4.3.
http://tools.ietf.org/html/rfc1952, 1996.

[59] IETF. RFC 2461: Neighbor discovery for ip version 6 (ipv6).
http://www.ietf.org/rfc/rfc2461.txt, 1998.

[60] IETF. RFC 2578: Structure of management information version 2 (smiv2). http://tools.ietf.org/html/rfc2578, 1999.

[61] IETF. RFC 3411 - 3418 simple network management protocol. http://tools.ietf.org/html/rfc3411, 2002.

[62] IETF. RFC 3927: Dynamic configuration of ipv4 link-local addresses. http://tools.ietf.org/html/rfc3927, 2005.

[63] IETF. RFC 4994: Transmission of ipv6 packets over ieee 802.15.4 networks. http://tools.ietf.org/html/rfc4944, 2007.

[64] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *MOBICOM*, pages 56–67. ACM, 2000.

[65] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva. Directed diffusion for wireless sensor networking. volume 11, pages 2–16, 2003.

[66] International Electrotechnical Commission (IEC), TC65. IEC 61131: Programmable controllers - parts 1 to 8, 2001-2004.

[67] International Electrotechnical Commission (IEC), TC65. IEC 61499: Function blocks - parts 1 to 4, 2004-2005.

[68] IP500 Alliance. http://www.ip500.de/.

[69] ITU. x.680 information technology - abstract notation one (asn.1): Specification of basic notation.

[70] ITU. x.691 information technology - asn.1 encoding rules: Specification of packed encoding rules (per).

[71] ITU. x.694 information technology - asn.1 encoding rules: Mapping w3c xml schema definitions into asn.1.

[72] ITU. X.891 information technology - generic applications of asn.1: Fast infoset.

[73] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. S. Peh, and D. Rubenstein. Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with zebranet. *SIGOPS Oper. Syst. Rev.*, 36(5):96–107, 2002.

[74] Y. jung Oh, H. taek Ju, M. jung Choi, and J. W. Hong. Interaction translation methods for xml/snmp gateway. In *In Proc. 13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, pages 54–65. Springer, 2002.

[75] Juniper Networks. JUNOScriptAPI, http://www.juniper.net/support/xml/junoscript/index.html.

[76] S. Käbisch, D. Peintner, J. Heuer, and H. Kosch. Efficient and flexible xml-based data-exchange in microcontroller-based sensor actor networks. In *Proceedings of the 2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops*, WAINA '10, pages 508–513, Washington, DC, USA, 2010. IEEE Computer Society.

[77] S. Käbisch, D. Peintner, J. Heuer, and H. Kosch. Xml-based web service generation for microcontroller-based sensor actor networks. In *8th IEEE International Workshop onFactory Communication Systems (WFCS)*, pages 181 –184, 2010.

[78] J. M. Kahn, R. H. Katz, Y. H. Katz, and K. S. J. Pister. Emerging challenges: Mobile networking for "smart dust". *Journal of Communications and Networks*, 2:188–196, 2000.

[79] V. Kawadia and P. R. Kumar. A cautionary perspective on cross layer. *Wireless Communications Magazine, IEEE*, 12:3–11, 2005.

[80] S. Kindermann, M. Stockhause, and K. Ronneberger. Intelligent data networking for the earth system science community. In *German e-Science Conference 2007*, Baden-Baden, May 2007.

[81] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[82] A. G. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[83] T. Klie and F. Strauß. Integrating snmp agents with xml-based management systems. *Communications Magazine, IEEE*, 42(7):76 – 83, july 2004.

[84] R. Kuntschke. *Network-Aware Optimization in Distributed Data Stream Management Systems.* PhD thesis, Technische Universität München, 2008.

[85] kXML. http://www.trantor.de/wbxml/.

[86] E. A. Lee, S. Neuendorffer, and M. J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, 12:231–260, 2003.

[87] W. L. Lee, A. Datta, and R. Cardell-Oliver. *Handbook of Mobile Ad Hoc and Pervasive Communications*, chapter Network management in wireless sensor networks. Scientific Publishers, USA.

[88] W. L. Lee, A. Datta, and R. Cardell-Oliver. Fleximac: A flexible tdma-based mac protocol for fault-tolerant and energy-efficient wireless sensor networks. In *Proc. IEEE ICON Conf.*, 2006.

[89] W. L. Lee, A. Datta, and R. Cardell-oliver. Winms: Wireless sensor network-management system, an adaptive policy-based management for wireless sensor networks. Technical report, UWA-CSSE-06-001, The University of Western Australia, 2006.

[90] P. Levis and D. Culler. Maté: a tiny virtual machine for sensor networks. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 85–95, New York, NY, USA, 2002. ACM.

[91] P. Levis, N. Lee, M. Welsh, and D. Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *Proceedings of the 1st international conference on Embedded networked sensor systems*, SenSys '03, pages 126–137, New York, NY, USA, 2003. ACM.

[92] Y. Y. Lim, M. Messina, F. Kargl, L. Ganguli, M. Fischer, and T. Tsang. Snmp proxy for wireless sensor network. In *ITNG '08: Proceedings of the Fifth International Conference on Information Technology: New Generations*, pages 738–743, Washington, DC, USA, 2008. IEEE Computer Society.

[93] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. In *IN OSDI*, 2002.

[94] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: An acquisitional query processing system for sensor networks. *ACM Trans. Database Syst*, 30:2005, 2005.

[95] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97, New York, NY, USA, 2002. ACM.

[96] R. Mangharam and M. Pajic. Embedded virtual machines for robust wireless control systems. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems Workshops*, ICDCSW '09, pages 38–43, Washington, DC, USA, 2009. IEEE Computer Society.

[97] R. Mangharam, M. Pajic, and S. Sastry. Demo abstract: Embedded virtual machines for wireless industrial automation. In *IPSN '09: Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, pages 413–414, Washington, DC, USA, 2009. IEEE Computer Society.

[98] D. Marinakis, P. Giguère, and G. Dudek. Learning network topology from simple sensor data. In *CAI '07: Proceedings of the 20th conference of the Canadian Society for Computational Studies of Intelligence on Advances in Artificial Intelligence*, pages 417–428, Berlin, Heidelberg, 2007. Springer-Verlag.

[99] M. Maróti, P. Völgyesi, S. Dóra, B. Kusý, A. Nádas, A. Lédeczi, G. Balogh, and K. Molnár. Radio interferometric geolocation. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 1–12, New York, NY, USA, 2005. ACM.

[100] P. J. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel. Flexcup: A flexible and efficient code update mechanism for sensor networks. In *In Proceedings of the Third European Workshop on Wireless Sensor Networks (EWSN 2006*, pages 212–227, 2006.

[101] P. J. Marrón, D. Minder, A. Lachenmann, and K. Rothermel. TinyCubus: An Adaptive Cross-Layer Framework for Sensor Networks. *it - Information Technology*, 47(2):87–97, April 2005.

[102] J.-P. Martin-Flatin. *Web-Based Management of IP Networks and Systems.* John Wiley & Sons, Inc., New York, NY, USA, 2002.

[103] C. J. Merlin. *Adaptability in Wireless Sensor Networks Through Cross-Layer Protocols and Architectures.* PhD thesis, Department of Electrical and Computer Engineering, University of Rochester, Rochester NY, 2009.

[104] D. Moore, J. Leonard, D. Rus, and S. Teller. Robust distributed network localization with noisy range measurements. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 50–61, New York, NY, USA, 2004. ACM.

[105] R. Müller, G. Alonso, and D. Kossmann. A virtual machine for sensor networks. *SIGOPS Oper. Syst. Rev.*, 41(3):145–158, 2007.

[106] M. Nelson, B.-H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 25–25, Berkeley, CA, USA, 2005. USENIX Association.

[107] NXP Semiconductors. The i2c-bus specification, version 3.0. http://www.nxp.com/acrobat_download/usermanuals/UM10204_3.pdf, 2007.

[108] OASIS. RELAX NG specification. http://relaxng.org/spec-20011203.html, 2001.

[109] OASIS. Oasis reference model for service oriented architecture 1.0, October 2006.

[110] OASIS. Open building information exchange. http://www.oasis-open.org/committees/download.php/21812/obix-1.0-cs-01.pdf, 2006.

[111] OASIS. Web services resource 1.2 (ws-resource). http://docs.oasis-open.org/wsrf/wsrf-ws_resource-1.2-spec-os.pdf, 2006.

[112] OASIS. Web services business process execution language version 2.0. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html, 2007.

[113] Objective Systems. Asn.1 viewer/editor. http://www.obj-sys.com/asn1-viewer.php.

[114] H. Oi. Hardware support for a wireless sensor network virtual machine. In *MOBILWARE '08: Proceedings of the 1st international conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications*, pages 1–5, ICST, Brussels, Belgium, Belgium, 2007. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[115] Open Geospatial Consortium (OGC). http://www.opengeospatial.org/standards/.

[116] Open Mobile Alliance. Wap binary xml content format specification (WBXML, version 1.3, July 2001.

[117] C. Peltz. Web services orchestration and choreography. *Computer*, 36:46–52, 2003.

[118] A. Pras, T. Drevers, R. V. D. Meent, and D. Quartel. Comparing the performance of snmp and web services-based management. In *ETransactions on network and service management*, 2004.

[119] Y. Z. Q. Wang and L. Cheng. Reprogramming wireless sensor networks: Challenges and approaches. *IEEE Network Magazine*, 20(3):48– 55, May-June 2006.

[120] H. Qi, Y. Xu, and X. Wang. Mobile-agent-based collaborative signal and information processing in sensor networks. *Proceedings of the IEEE*, 91(8):1172 – 1183, aug. 2003.

[121] M. Quaritsch, M. Kreuzthaler, B. Rinner, H. Bischof, and B. Strobl. Autonomous multicamera tracking on embedded smart cameras. *EURASIP J. Embedded Syst.*, 2007(1):35–35, 2007.

[122] M. A. Razzaque, S. Dobson, and P. Nixon. Cross-layer architectures for autonomic communications. *J. Netw. Syst. Manage.*, 15(1):13–27, 2007.

[123] K. Römer and F. Mattern. The design space of wireless sensor networks, 2004.

[124] L. B. Ruiz. *MANNA: A Management Architecture for Wireless Sensor Networks.* PhD thesis, Federal University of Minas Gerais, Belo Horizonte, MG, Brazil, 2003.

[125] L. B. Ruiz, J. M. Nogueira, and A. A. F. Loureiro. Manna: a management architecture for wireless sensor networks. In *Communications Magazine, IEEE*, volume 41, pages 116–125, 2003.

[126] L. B. Ruiz, I. G. Siqueira, L. B. e. Oliveira, H. C. Wong, J. M. S. Nogueira, and A. A. F. Loureiro. Fault management in event-driven wireless sensor networks. In *MSWiM '04: Proceedings of the 7th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems*, pages 149–156, New York, NY, USA, 2004. ACM.

[127] S. J. Russell and P. Norvig. *Artificial Intelligence: a modern approach.* Prentice Hall, 2nd international edition edition, 2003.

[128] P. Santi. Topology control in wireless ad hoc and sensor networks. *ACM Comput. Surv.*, 37(2):164–194, 2005.

[129] A. Scholz, C. Buckl, I. Gaponova, S. Sommer, A. Knoll, A. Kemper, J. Heuer, and A. Schmitt. An adaptive soa for embedded networks. In *INDIN'09: 7th IEEE International Conference on Industrial Informatics.* IEEE, 2009.

[130] A. Scholz, C. Buckl, S. Sommer, A. Kemper, A. Knoll, J. Heuer, and A. Schmitt. esoa - soa für eingebettete netze. *ECEASST*, 17, 2009.

[131] A. Scholz, I. Gaponova, S. Sommer, A. Kemper, A. Knoll, C. Buckl, J. Heuer, and A. Schmitt. Efficient communication in control-oriented embedded networks. In *Proceedings of the 14th IEEE international conference on Emerging technologies & factory automation*, ETFA'09, pages 132–139, Piscataway, NJ, USA, 2009. IEEE Press.

[132] A. Scholz, S. Sommer, C. Buckl, G. Kainz, A. Kemper, A. Knoll, J. Heuer, and A. Schmitt. Towards an adaptive execution of applications in heterogeneous embedded networks. In *Software Engineering for Sensor Network Applications (SESENA 2010).* ACM/IEEE, 2010.

[133] J. Schönwälder, A. Pras, and J.-P. Martin-Flatin. On the future of internet management technologies. *IEEE Communications Magazine*, 41(10):90–97, 2003.

[134] P. Shafer. Xml-based network management.
http://www.juniper.net/solutions/literature/white_papers/200017.pdf.

[135] R. C. Shah and J. M. Rabaey. Energy aware routing for low energy ad hoc sensor networks. In *IEEE Wireless Communications and Networking Conference (WCNC)*, March 2002.

[136] S. Shenker, S. Ratnasamy, B. Karp, R. Govindan, and D. Estrin. Data-centric storage in sensornets. *SIGCOMM Comput. Commun. Rev.*, 33(1):137–142, 2003.

[137] V. Shnayder, M. Hempstead, B.-r. Chen, G. W. Allen, and M. Welsh. Simulating the power consumption of large-scale sensor network applications. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, SenSys '04, pages 188–200, New York, NY, USA, 2004. ACM.

[138] M. L. Shooman. *Reliability of Computer Systems and Networks: Fault Tolerance, Analysis, and Design.* John Wiley & Sons, 2002.

[139] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java on the bare metal of wireless sensor devices: the squawk java virtual machine. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 78–88, New York, NY, USA, 2006. ACM.

[140] R. Sollacher, C. Niedermeier, N. Vicari, and M. Osipov. Towards a service oriented architecture for wireless sensor networks in industrial applications. In *13th IFAC Symposium on Information Control Problems in Manufacturing (INCOM 2009)*, 2009.

[141] S. Sommer, A. Scholz, C. Buckl, A. Knoll, A. Kemper, J. Heuer, and A. Schmitt. Towards the internet of things: Integration of web services and field level devices. In *FITS '09: International Workshop on the Future Internet of Things and Services - Embedded Web Services for Pervasive Devices*, 2009.

[142] S. Sommer, A. Scholz, I. Gaponova, A. Knoll, A. Kemper, C. Buckl, G. Kainz, J. Heuer, and A. Schmitt. Service migration scenarios for embedded networks. *Advanced Information Networking and Applications Workshops, International Conference on*, 0:502–507, 2010.

[143] H. Song, D. Kim, K. Lee, and J. Sung. Upnp-based sensor network management architecture. In *Proc. ICMU*, 2005.

[144] B. Srivastava and J. Koehler. Web service composition - current solutions and open problems. In *In: ICAPS 2003 Workshop on Planning for Web Services*, pages 28–35, 2003.

[145] V. Srivastava and M. Motani. Cross-layer design: a survey and the road ahead. *Communications Magazine, IEEE*, 43(12):112–119, 2005.

[146] W. Stallings. *SNMP, SNMPv2, and CMIP - The Pracitcal Guide to Network-Management Standards.* Addison-Wesley Publishing Company, 1993.

[147] F. Strauß and T. Klie. Towards xml oriented internet management. In *in Proc. 8th IFIP/IEEE International Symposium on Integrated Network Management*, pages 505–518. Kluwer Academic Publishers, 2003.

[148] C. P. Sun. Ad-hoc on-demand distance vector routing. In *In Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, 1997.

[149] Sun Microsystems Laboratories. Sun spot world.
http://www.sunspotworld.com/.

[150] A. I. Sundararaj and P. A. Dinda. Towards virtual networks for virtual machine grid computing. In *VM'04: Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium*, pages 14–14, Berkeley, CA, USA, 2004. USENIX Association.

[151] H. taek Ju, M. jung Choi, S. Han, Y. Oh, J. hyuk Yoon, H. Lee, and J. W. Hong. An embedded web server architecture for xml-based network management. In *Management, Proc. IEEE/IFIP Network Operations and Management Symposium (NOMS 2002*, pages 1–14, 2002.

[152] A. Talevski, S. Carlsen, and S. Petersen. Research challenges in applying intelligent wireless sensors in the oil, gas and resources industries. In *Industrial Informatics, 2009. INDIN 2009. 7th IEEE International Conference on*, pages 464–469, June 2009.

[153] F. Teraoka and M. Tokoro. Host migration transparency in ip networks: the vip approach. *SIGCOMM Comput. Commun. Rev.*, 23(1):45–65, 1993.

[154] G. Tolle and D. Culler. Design of an application-cooperative management system for wireless sensor networks. In *Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on*, pages 121–132, February 2005.

[155] W3C. Document object model (DOM). http://www.w3.org/DOM/.

[156] W3C. Semantic annotations for WSDL working groups.
http://www.w3.org/2002/ws/sawsdl/.

[157] W3C. Web service semantics - WSDL-S.
http://www.w3.org/Submission/WSDL-S/.

[158] W3C. Hypertext transfer protocol - HTTP/1.1.
http://tools.ietf.org/html/rfc2616, 1999.

[159] W3C. Web services description language (WSDL) 1.1.
http://www.w3.org/TR/wsdl, 2001.

[160] W3C. Web service choreography interface (WSCI) 1.0.
http://www.w3.org/TR/wsci, 2002.

[161] W3C. Web service management: Service life cycle.
http://www.w3.org/TR/wslc/, 2004.

[162] W3C. XML Schema part 1: Structures second edition.
http://www.w3.org/TR/xmlschema-1/, 2004.

[163] W3C. XML Schema part 2: Datatypes second edition.
http://www.w3.org/TR/xmlschema-2/, 2004.

[164] W3C. Web services choreography description language version (WS-CDL) 1.0.
http://www.w3.org/TR/ws-cdl-10/, 2005.

[165] W3C. Simple object access protocol (SOAP).
http://www.w3.org/TR/soap/, 2007.

[166] B. Warneke, M. Last, B. Liebowitz, and K. S. Pister. Smart dust: Communicating with a cubic-millimeter computer. *Computer*, 34:44–51, 2001.

[167] J. Wiklander, J. Eliasson, A. Kruglyak, P. Lindgren, and J. Nordlander. Enabling component-based design for embedded real-time software. *JCP*, 4(12):1309–1321, 2009.

[168] R. Winter and J. Schiller. Crosstalk: A data dissemination-based crosslayer architecture for mobile ad-hoc networks. In *in Proceedings of ASWN*, 2005.

[169] G. Wittenburg and J. Schiller. A survey of current directions in service placement in mobile ad-hoc networks. *Pervasive Computing and Communications, IEEE International Conference on*, 0:548–553, 2008.

[170] XimpleWare. VTD-XML. http://vtd-xml.sourceforge.net/.

[171] J. Yick, G. Pasternack, B. Mukherjee, and D. Ghosal. Placement of network services in a sensor network. *Int. J. Wire. Mob. Comput.*, 1:101–112, February 2006.